Symbolic Learning for Adaptive Agents

Joshua Cole
Computer Sciences Laboratory
The Australian National University
Email: jcole@discus.anu.edu.au

John Lloyd
Computer Sciences Laboratory
The Australian National University
Email: jwl@discus.anu.edu.au

Kee Siong Ng
Computer Sciences Laboratory
The Australian National University
Email: kee.siong@discus.anu.edu.au

***Abstract*— This paper investigates an approach to designing and building adaptive agents. The main contribution is the use of a symbolic machine learning system for approximating the policy and Q functions that are at the heart of the agent. Under the assumption that sufficient knowledge of the application domain is available, it is shown how this knowledge can be provided to the agent in the form of symbolic hypothesis languages for the policy and Q functions, and the advantages of such an approach. A series of experiments concerning the performance of an agent employing this architecture in the blocks world domain is presented and some general conclusions drawn.**

I. Introduction

In recent years, there has been a substantial increase in interest in designing, building, and deploying agents in a variety of application domains. In particular, a number of possible approaches to the architecture of agents have been explored in great detail. (For an excellent overview, see [10].) Nevertheless, it is clear that many issues are still unsettled and that considerable work remains to be done. One general issue that is still open is how the reasoning and learning components of an agent might be seamlessly integrated. At the moment, reasoning issues and learning issues tend to be discussed in separate research communities, and their integration has only rarely been addressed.

This general issue of the integration of reasoning and learning capabilities provides the overall motivation for our work on agent architectures. However, in this paper, we concentrate specifically on learning issues. In particular, this paper is concerned with the issue of designing agent architectures that are adaptive, that is, agents using such architectures have the ability to perform well even if the environment in which they are operating changes.

Our general approach is based on the well established concepts of Markov decision processes [9] and reinforcement learning [2], [12]. More specifically, we take up the approach of relational reinforcement learning by Džeroski *et al* in [4] and [5]. The key idea of this approach is to use a symbolic (that is, logical) machine learning system to approximate the policy and Q functions instead of the more usual approach of using a (non-symbolic) learning system such as a neural network [2]. Using a *symbolic* learning system turns out to be a very fruitful idea. To begin with, it provides a convenient way of incorporating domain knowledge into the agent in the form of hypothesis languages for the policy and Q functions. Not surprisingly, this knowledge can be used to greatly reduce the size of the search problems associated with reinforcement learning. Furthermore, it provides policy and Q functions that are in symbolic form; therefore, they are comprehensible and explicitly manipulable. So, for example, the agent is able to explain to a user why it behaved in a certain way.

Our view is that having a symbolic form of the policy function of an agent is key to the seamless integration of reasoning and learning capabilities, and our future work beyond this paper will explore this idea. But note carefully that an important assumption is being made here which is that the designer of the agent does have appropriate knowledge of the application domain. We believe that in many agent applications, this knowledge is indeed available to be exploited. Furthermore, the more knowledge that is available the better. In general, in the context of agents, we regard learning as a 'last resort' technology. The intuition is that when building agents as much domain knowledge as possible should be built in and learning techniques should only be used when this knowledge is incomplete. Important situations when learning becomes essential are where an agent is deployed in a dynamic environment and where a 'generic' agent needs to adapt to the behaviour pattern of a particular user.

An outline of this paper is as follows. The next section provides an overview of our approach to adaptive agent architectures. Section III introduces ALKEMY, the symbolic learning system used in the agent architecture. Section IV describes the blocks world domain in which the experiments were carried out. Section V presents the experiments and results. Finally, Section VI gives some conclusions.

II. An Adaptive Agent Architecture

The agent architecture is based on Markov decision processes [9]. We assume discrete time, so there is a set T of time steps of the form $\{0, 1, 2, \ldots\}$.

Definition 1. A *Markov decision process* consists of the following:

1) a finite set S_t of states, for each $t \in T$.
2) a finite set A_t of actions, for each $t \in T$.
3) for each state $s_t \in S_t$ and each action $a_t \in A_t$, a transition probability distribution $p_t(\cdot \mid s_t, a_t)$, for each $t \in T$.
4) a reward function $r_t : S_t \times A_t \to \mathbb{R}$, for each $t \in T$.

Note that the states, actions, transition probability distribution, and the reward function are all indexed by the time t. At time t, the agent perceives the current state to be

$s_t \in S_t$. It then chooses from amongst the legal actions an action $a_t \in A_t$ and performs it. The next percept from the environment gives the reward $r_t(s_t, a_t)$ to the agent and the next state is $s_{t+1} \in S_{t+1}$ with probability $p_t(s_{t+1} \mid s_t, a_t)$. It is assumed that the agent does not know the transition probability distribution $p_t(\cdot \mid s, a)$ nor the reward function r_t.

A solution to a Markov decision process is a *policy sequence* of the form $\pi = [\pi_0, \pi_1, \pi_2, \ldots]$, where each π_t is a function called a *policy* from S_t to A_t. Given the current state $s_t \in S_t$ at time t, the action prescribed by the policy sequence is $a_t = \pi_t(s_t)$. To find a good policy sequence, we need a way to evaluate their quality. We define the *discounted total reward* $V^\pi(s)$ by following a policy sequence π from an arbitrary initial state s as

$$V^\pi(s) \equiv \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t(s_t, a_t) \mid \pi, s_0 = s\right],$$

where γ is the discount factor for which $0 \leq \gamma < 1$. Given an initial state s, an optimal policy sequence π_s^* can then be defined as

$$\pi_s^* \equiv \arg\max_\pi V^\pi(s).$$

Thus the agent attempts to find a policy sequence that maximises its discounted total reward given some initial state.

We next present a preliminary version of an adaptive agent architecture based on these ideas. The underlying adaptation algorithm is a form of Q-learning [13] with function approximation [2]. The approach taken is motivated by the work on relational reinforcement learning in [4] and [5]. We represent states, actions, and value functions symbolically. For function approximation, we use an incremental version of a higher-order decision-tree learner called ALKEMY, more details of which are given in Section III.

An important innovation introduced in [5], which we adopt in our architecture, is P-learning. The basic idea is that one can associate with a Q function a boolean-valued policy function P defined as follows:

if $a = \arg\max_{a'} Q(s, a')$ then $P(s, a) = \top$ else $P(s, a) = \bot$.

The main observation is that the Q function explicitly encodes the path length to a goal from a given state, and this is complex and specific to particular worlds. By computing P from Q, one obtains a more compact representation of the policy, and this has obvious implications for the stability and predictive power of the policy. Furthermore, by carefully crafting the hypothesis languages for P and Q, it is possible to achieve generalisation across problem instances. In other words, given a class of tasks of a similar nature, one can pick an instance from the class, train the agent on that instance to obtain a P function via Q-learning, and then (re)use P as a generic policy to solve other problems in the class. This phenomenon will be elaborated further in Section V.

Figure 1 shows an overview of the proposed agent architecture. Following the tradition of BDI architectures, the agent

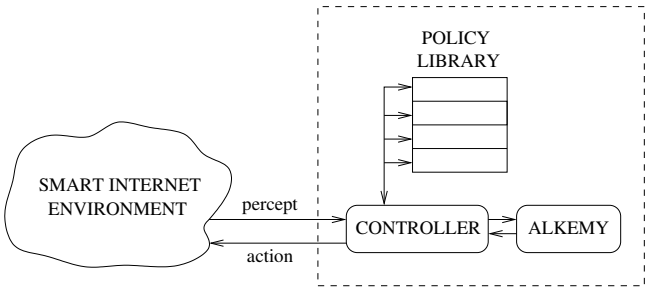


Fig. 1. The agent architecture

is equipped with a policy library. There are one or more policies for each kind of task that the agent can perform. Each policy is encoded using an ALKEMY decision tree, coupled with an hypothesis search space and constraints on how the tree can be modified. The agent interacts with the world in the usual manner, by perceiving the environment and performing actions. From those interactions, training examples are generated which ALKEMY uses to update the policies in the library to improve its performance. The revision of policies is guided by the learning of Q functions, which provide vital information about the quality of policies otherwise unavailable to the agent.

Figure 2 gives the agent algorithm. Given a task $\mathcal{T}$, the agent selects from the library a policy that matches the problem and uses it to initialise P and Q. It then goes into a loop, performing actions by a trade-off between exploitation of P and exploration of the state space, collecting rewards and observing the effects of its actions. Training examples are generated to update the P and Q functions in each iteration. Note that action selection and the two update functions are parameters in the algorithm. Depending on the situation, one may prefer to do more or less exploration. Further, one may choose different ways and frequencies of updating the P and Q functions. In Section V, we explore different instantiations of these functions for different scenarios.

The proposed scheme is attractive for two reasons. First, it provides a convenient mechanism to encode background knowledge about the problem domain, perhaps the most important factor in the engineering of agents that actually do useful things. Second, from a design point of view, the scheme is aesthetically pleasing because the architecture encompasses as special cases PRS-like systems [6] with plan libraries (plans are just policy functions that never change) and standard Q-learning with function approximation for which the set of states, set of actions, transition probability distributions, and reward functions stay the same for all $t \in T$.

We note that BDI-like reasoning can be incorporated to compute the most important task to be executed at any point in time. It is also likely that more advanced reinforcement learning methods can be employed in the architecture in place of Q-learning. These extensions are currently under investigation.

Algorithm $Agent(\mathcal{T})$
input: $\mathcal{T}$, a task;

$t :=$ current time;
$s_t :=$ current state;
$P_t := policyLibrary[\mathcal{T}].P$;
$Q_t := policyLibrary[\mathcal{T}].Q$;
while $\mathcal{T}$ not done
  $a_t := selectAction(s_t, P_t)$;
  perform a_t;
  observe $r = r_t(s_t, a_t)$ and s_{t+1};
  $x := ((s_t, a_t),\ r + \gamma \max_{a \in A_{t+1}} Q_t(s_{t+1}, a))$;
  $Q_{t+1} := updateQ(Q_t, \{x\})$;
  $X := \emptyset$;
  forall $a \in A_t$ **do**
    if $a = \arg\max_{a' \in A_t} Q_{t+1}(s_t, a')$
    then $x := ((s_t, a), \top)$;
    else $x := ((s_t, a), \bot)$;
    $X := X \cup \{x\}$;
  $P_{t+1} := updateP(P_t, X)$;
  $t := t + 1$;
$policyLibrary[\mathcal{T}].P := P_t$;
$policyLibrary[\mathcal{T}].Q := Q_t$;

Fig. 2. The agent algorithm

III. A Symbolic Learning System

Decision-tree learning systems are based on the following intuitively appealing idea. To build a classifier from a set of examples, find a criterion that partitions the examples into two sets which are purer in the distribution of classes that they contain than the original set; apply this process recursively on the child nodes until the leaf nodes of the tree are sufficiently pure; and then use the resulting decision tree as the induced classifier. Learning regression functions is based on a similar intuition. Decision-tree learning is one of few learning methods that provides comprehensible hypotheses and this explains the concentration on that approach here.

The learning system employed here, called ALKEMY, is a classification and regression system. Information about ALKEMY, and the system itself, is available at [1]. The theoretical foundations of ALKEMY are presented in [7]. Here we only provide sufficient detail in order to understand the crucial role that ALKEMY plays in the agent architecture. The main topic is therefore that of hypothesis languages and how predicate rewrite systems are used to specify these. Predicates in hypothesis languages are constructed incrementally by composing more basic functions, so the development begins with the definition of composition.

Composition is handled by the (reverse) composition function

$$\circ : (a \to b) \to (b \to c) \to (a \to c)$$

defined by

$$((f \circ g)\ x) = (g\ (f\ x)).$$

Predicates are built up by composing transformations, which are defined as follows.

Definition 2. A *transformation* f is a function having a signature of the form

$$f : (\varrho_1 \to \Omega) \to \cdots \to (\varrho_k \to \Omega) \to \mu \to \sigma,$$

where any parameters in $\varrho_1, \ldots, \varrho_k$ and σ appear in μ, and $k \geq 0$. (Here Ω is the type of the booleans.) The type μ is distinguished and is called the *source* of the transformation, while the type σ is called the *target* of the transformation. The number k is called the *rank* of the transformation.

Example 1. The transformation

$$\wedge_n : (a \to \Omega) \to \cdots \to (a \to \Omega) \to a \to \Omega$$

defined by

$$\wedge_n\ p_1 \ldots p_n\ x = (p_1\ x) \wedge \cdots \wedge (p_n\ x),$$

where $n \geq 2$, provides a 'conjunction' with n conjuncts.

Example 2. Each projection

$$proj_i : a_1 \times \cdots \times a_n \to a_i$$

defined by

$$proj_i\ (t_1, \ldots, t_n) = t_i,$$

for $i = 1, \ldots, n$, is a transformation of rank 0.

Example 3. There are two fundamental transformations $top : a \to \Omega$ and $bottom : a \to \Omega$ defined by $top\ x = \top$ and $bottom\ x = \bot$, for each x. Each of top and $bottom$ is a constant predicate, with top being the weakest predicate on the type a and $bottom$ being the strongest.

Example 4. Let μ be a type and suppose that $A : \mu$, $B : \mu$, and $C : \mu$ are constants. Then, corresponding to A, one can define a transformation

$$(= A) : \mu \to \Omega$$

by

$$((= A)\ x) = x = A,$$

with analogous definitions for $(= B)$ and $(= C)$. Similarly, one can define the transformation

$$(\neq A) : \mu \to \Omega$$

by

$$((\neq A)\ x) = x \neq A.$$

Example 5. Consider a type such as Int (the type of the integers) which has various order relations defined on it. Then, for any integer N, one can define the transformation

$$(< N) : Int \to \Omega$$

by

$$((< N)\ m) = m < N.$$

In a similar way, one can define the transformations $(> N)$, $(\geq N)$, and $(\leq N)$.

Transformations are used to define a particular class of predicates, called standard predicates, that are used in hypothesis languages.

Definition 3. A *standard predicate* is a term of the form

$$(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n}),$$

where f_i is a transformation of rank k_i ($i = 1, \ldots, n$), the target of f_n is Ω, p_{i,j_i} is a standard predicate ($i = 1, \ldots, n$, $j_i = 1, \ldots, k_i$), $k_i \geq 0$ ($i = 1, \ldots, n$) and $n \geq 1$.

Example 6. If p, q, and r are standard predicates (having appropriate type) and $\neg : \Omega \to \Omega$ is negation, then $(\wedge_3\ p\ q\ r) \circ \neg$ is a standard predicate.

Now we can very informally define a predicate rewrite system. A *predicate rewrite* is an expression of the form

$$p \rightarrowtail q,$$

where p and q are standard predicates. The predicate p is called the *head* and q is the *body* of the rewrite. A *predicate rewrite system* is a finite set of predicate rewrites. One should think of a predicate rewrite system as a kind of grammar for generating a particular hypothesis language. Roughly speaking, this works as follows. Starting from the weakest predicate top, all predicate rewrites that have top (of the appropriate type) in the head are selected to make up child predicates that consist of the bodies of these predicate rewrites. Then, for each child predicate and each redex in that predicate, all child predicates are generated by replacing each redex by the body of the predicate rewrite whose head is identical to the redex. This generation of predicates continues to produce the hypothesis language. In [7], methods of efficiently generating the predicates for an hypothesis language are explored.

In the experiments, we use an on-line version of ALKEMY. The algorithms and their analyses are given in [8]. The basic idea is that we maintain a fixed window of a certain size. As each new example is encountered, it is inserted into the current tree using a function called $addExample$. If the window is full, the oldest example is removed from the tree using a function called $removeExample$. The two functions have the following property. They employ some sufficient optimality-testing conditions to check whether parts of the tree have become potentially sub-optimal according to the tree-building measures as a result of the update, and mark as dirty all those branches that need to be re-examined. There is a function $retrain$ that the agent can call to rebuild all the dirty branches of a tree. The function $retrain$ has the property that the tree computed is exactly identical to the tree that would be produced by the batch algorithm using the examples in the current window. For that reason, we say the on-line algorithm is *lossless* with respect to the batch algorithm.

IV. BLOCKS WORLD DOMAIN

The experiments were carried out in the blocks world domain, which serves as a simple, yet sufficiently rich, domain in which to carry out experiments and draw lessons regarding general architectural issues.

Here are some declarations suitable for this domain.

$$B_0, B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, Floor : Object$$
$$Stack = List\ Object$$
$$World = \{Stack\}$$
$$OnState = Object \times Object$$
$$Intention = \{OnState\}$$
$$Action = Object \times Stack$$
$$State = World \times Intention$$
$$Individual = State \times Action.$$

The number of blocks in the world varies from time to time, but we never use more than ten blocks. A blocks world is modelled as a set of stacks of blocks and corresponds to the agent's beliefs, in the BDI sense of belief. The agent's intentions are modelled as a set of on-states, where an *on-state* is a pair of objects, the first of which is intended to be immediately on top of the second. An action specifies that some block that is clear should be put on top of some stack. The empty stack is allowed and is another representation of the floor, so that moving a block to the empty stack is actually moving the block to the floor. A state is a pair consisting of a blocks world and a set of intentions. These intentions should be interpreted in the BDI sense: the agent intends to achieve these intentions. In the experiments, the intentions are provided externally by percepts that the agent immediately accepts as intentions. Finally, an individual is a pair consisting of a state and an action. Several functions below whose domain is the set of individuals are the subject of learning.

A block in the world component of an individual is *misplaced* if its position in the world is inconsistent with the on-states specified in the intention component of the individual. An action is *constructive* if after the move of the block specified by the action, neither the block nor any block underneath it is misplaced and the move achieves an on-state in the intention. Once a block has been moved constructively, it need not move again in the course of achieving the intention. An individual is *deadlocked* if no constructive move is possible with respect to the world and intention components of that individual.

Each Q_t function has signature

$$Q_t : Individual \to \mathbb{R}.$$

The hypothesis language for each Q_t function has a single domain-specific transformation that has signature

$$estimatedPathLength : Individual \rightarrow Int$$

and is defined as follows. Suppose, in the individual, the action is to move block A on top of some stack that has block B at the top. Then the value of $estimatedPathLength$ for that individual is given by

$$\begin{aligned}
&2 \times \text{number of misplaced blocks in the world} \\
&+ \begin{cases} -1 & \text{if } A \text{ is intended to be on } B \\ +1 & \text{if } A \text{ is intended to be on } C(\neq B) \text{ or} \\ & C(\neq A) \text{ is intended to be on } B \\ 0 & \text{otherwise} \end{cases} \\
&+ \begin{cases} -1 & \text{if } A \text{ is misplaced} \\ 0 & \text{otherwise} \end{cases} \\
&+ \begin{cases} +1 & \text{if } B \text{ is misplaced} \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}$$

The function $estimatedPathLength$ is intended to provide an estimate of the shortest path to a goal (that is, a state satisfying the intention) from the state that results by applying the action to the current state. Such an estimate is needed to approximate each Q_t function. Note that using a transformation such as $estimatedPathLength$ implies that the agent has some limited knowledge of the effects of its actions. Here is the predicate rewrite system for the Q_t hypothesis language.

$$\begin{aligned}
&top \rightarrowtail estimatedPathLength \circ top \\
&top \rightarrowtail (=0) \\
&top \rightarrowtail (=1) \\
&\quad \vdots \\
&top \rightarrowtail (=21) \\
&top \rightarrowtail (=22).
\end{aligned}$$

Each Q_t function for a world in which there are N blocks, represented as an ALKEMY regression tree, typically has the following general structure.

$$\begin{aligned}
&Q_t\ x = \\
&\quad if\ estimatedPathLength \circ (=0)\ x \\
&\quad then\ q_0 \\
&\quad else\ if\ estimatedPathLength \circ (=1)\ x \\
&\qquad then\ q_1 \\
&\qquad\ \vdots \\
&\qquad else\ if\ estimatedPathLength \circ (=2N+1)\ x \\
&\qquad\quad then\ q_{2N+1} \\
&\qquad\quad else\ q_{otherwise},
\end{aligned}$$

where q_0, q_1, and so on, are the various regression values.

Now we turn attention to the policy function. Instead of directly learning a policy function

$$\pi_t : State \rightarrow Action,$$

a policy *relation* having signature

$$policy_t : Individual \rightarrow \Omega$$

is learned. To determine the policy function π_t from $policy_t$, we proceed as follows. Let s be a state. For each action a, determine the value of $policy_t\ (s, a)$. If there is at least one a for which $policy_t\ (s, a) = \top$, choose an a arbitrarily amongst all such a. Otherwise, choose a arbitrarily amongst all a. Depending on the nature of $policy_t$, the function π_t may thus be non-deterministic.

The hypothesis language for the policy relation requires a number of domain-specific transformations.

The transformation

$$extractAction : Individual \rightarrow Object \times Stack \times Individual$$

takes an individual as input and returns the triple consisting of the block in the action, the stack in the action, and the individual.

The transformation

$$projA : Object \times Stack \times Individual \rightarrow Object \times Individual$$

takes as input a triple consisting of a block, a stack, and an individual, and returns the pair consisting of the block and the individual.

The transformation

$$projB : Object \times Stack \times Individual \rightarrow Stack \times Individual$$

takes as input a triple consisting of a block, a stack, and an individual, and returns the pair consisting of the stack and the individual.

The transformation

$$noMisplacedNotOnFloorBlock : Individual \rightarrow \Omega$$

takes as input an individual and returns true, if there is no block in the world component of the individual that is misplaced and not on the floor; and returns false, otherwise.

The transformation

$$isDeadlocked : Individual \rightarrow \Omega$$

takes as input an individual and returns true, if there is no constructive move possible in the world component of the individual; and returns false, otherwise.

The transformation

$$isMisplaced : Object \times Individual \rightarrow \Omega$$

takes as input a pair consisting of a block and an individual and returns true, if the block is misplaced in the world component of the individual; and returns false, otherwise.

The transformation

$$isFloor : Stack \times Individual \rightarrow \Omega$$

takes as input a pair consisting of a stack and an individual and returns true, if the stack is empty; and returns false, otherwise.

The transformation

$$isConstructive : Object \times Stack \times Individual \rightarrow \Omega$$

takes as input a triple consisting of a block, a stack, and an individual, and returns true if moving the block to the stack in the world component of the individual is constructive; and returns false, otherwise.

The transformation

$$achievesSingletonIntention : Object \times Stack \times Individual \rightarrow \Omega$$

takes as input a triple consisting of a block A, a stack B, and an individual, and returns true if the top of stack B is block C and the intention component of the individual is $\{(A, C)\}$; and returns false, otherwise.

The transformation

$$notContainIntentionBlock : Stack \times Individual \rightarrow \Omega$$

takes as input a pair consisting of a stack and an individual, and returns true if the stack does not contain any block appearing in the intention component of the individual; and returns false, otherwise.

The transformation

$$aboveIntentionBlock : Object \times Individual \rightarrow \Omega$$

takes as input a pair consisting of a block and an individual, and returns true if the block is above some block in the intention component of the individual; and returns false, otherwise.

Here is a predicate rewrite system for policy hypothesis languages. In the next section, particular subsets of this rewrite system will be used in various experiments.

$$top \rightarrowtail \wedge_2\ top\ top$$
$$top \rightarrowtail \wedge_3\ top\ top\ top$$
$$top \rightarrowtail extractAction \circ projA \circ top$$
$$top \rightarrowtail extractAction \circ projB \circ top$$
$$top \rightarrowtail extractAction \circ top$$
$$top \rightarrowtail noMisplacedNotOnFloorBlock$$
$$top \rightarrowtail isDeadlocked$$
$$top \rightarrowtail isMisplaced$$
$$top \rightarrowtail isFloor$$
$$top \rightarrowtail isConstructive$$
$$top \rightarrowtail achievesSingletonIntention$$
$$top \rightarrowtail notContainIntentionBlock$$
$$top \rightarrowtail aboveIntentionBlock.$$

We consider two policies, *US* and *GN1*, that were studied in [11], and two more restricted ones, *simple* and *one_intention*.

The *simple* policy either makes a constructive move or else moves a block to the floor. Here is the *simple* policy as an ALKEMY decision tree.

$$\begin{array}{l} policy_{simple}\ x = \\ \quad if\ extractAction \circ isConstructive\ x \\ \quad then\ \top \\ \quad else\ if\ \ extractAction \circ projB \circ isFloor\ x \\ \qquad then\ \top \\ \qquad else\ \bot. \end{array}$$

The *US* (Unstack-Stack) policy puts all misplaced blocks on the floor first and then builds the goal state by constructive moves. Here is the *US* policy as an ALKEMY decision tree.

$$\begin{array}{l} policy_{US}\ x = \\ \quad if\ \wedge_2\ (noMisplacedNotOnFloorBlock) \\ \qquad\qquad (extractAction \circ isConstructive)\ x \\ \quad then\ \top \\ \quad else\ if\ \wedge_2\ (extractAction \circ projA \circ isMisplaced) \\ \qquad\qquad (extractAction \circ projB \circ isFloor)\ x \\ \qquad then\ \top \\ \qquad else\ \bot. \end{array}$$

The *GN1* policy is as follows: if there is a constructive move, then do it; else arbitrarily choose a misplaced block and move it to the floor. Here is the *GN1* policy as an ALKEMY decision tree.

$$\begin{array}{l} policy_{GN1}\ x = \\ \quad if\ \ extractAction \circ isConstructive\ x \\ \quad then\ \top \\ \quad else\ if\ \wedge_3\ (extractAction \circ projA \circ isMisplaced) \\ \qquad\qquad (extractAction \circ projB \circ isFloor) \\ \qquad\qquad\qquad\qquad (isDeadlocked)\ x \\ \qquad then\ \top \\ \qquad else\ \bot. \end{array}$$

Finally, we explain the *one_intention* policy. Suppose the intention is to put block C onto block D, and the action is to move block A onto stack B. Then the *one_intention* policy is as follows: if the move achieves C on D, then do it; else if A is above C or D and B doesn't contain C or D, then do the move. Here is the *one_intention* policy as an ALKEMY decision tree.

$$\begin{array}{l} policy_{one\_intention}\ x = \\ \quad if\ \ extractAction \circ achievesSingletonIntention\ x \\ \quad then\ \top \\ \quad else\ if\ \wedge_3\ (extractAction \circ projA \circ aboveIntentionBlock) \\ \quad\ (extractAction \circ projB \circ notContainIntentionBlock) \\ \qquad\qquad (isDeadlocked)\ x \\ \qquad then\ \top \\ \qquad else\ \bot. \end{array}$$

V. Experiments and Results

This section discusses various experiments carried out in the blocks world domain, and the results obtained.

A. Experimental parameters

The general agent algorithm in Figure 2 has as parameters an action selection function $selectAction$, and two update functions $updateQ$ and $updateP$. These parameters are instantiated in the following experiments. Here is a general description of each function.

The $selectAction$ function stochastically selects a legal action a based on the current state s_t using the current policy relation $policy_t$ as described in Section IV. However, the agent can benefit from ignoring its current policy from time to time and exploring new regions of the state space. An *exploration factor* is employed to permit occasional random actions, irrespective of what the current policy says. An exploration factor of zero corresponds to 100% exploitation of the current policy.

The $updateQ$ function in these experiments collects training input after each action is performed but only requests ALKEMY to retrain to produce a new Q regression tree at the end of each episode or after 100 moves since the last retrain, whichever occurs first.

The $updateP$ function in these experiments provides a mechanism for delaying the collection of P training data in the early training phase when the Q regression tree is unlikely to provide a good prediction. A *training data delay* parameter controls this delay in the collection of P training data.

The $updateP$ function also permits the delaying of retraining of the P decision tree by two mechanisms: an initial delay and an accuracy measure. The first is a straightforward *retraining delay* parameter used to delay retraining of the P decision tree until sufficient data is available; the second requires further explanation.

The $updateP$ function gathers accuracy statistics about the incoming training data for P. It does this by first querying ALKEMY for its predicted P value for each (s_t, a) pair using the existing P decision tree. If the predicted P values tend to agree with the existing P training data, then the $updateP$ function delays requesting ALKEMY to retrain until an error threshold is breached. In most of the experiments that follow, this *error threshold* parameter is set at 15%.

Finally, like the $updateQ$ function, the $updateP$ function only ever requests ALKEMY to retrain the current P decision tree at the end of an episode or after 100 moves since the last retrain, whichever occurs first, provided that retraining has not been delayed for the reasons just mentioned.

The ALKEMY on-line learning window sizes used in all experiments were 200 training examples for the Q tree and 1000 training examples for the P tree (because there are about five times as many P training examples for each Q training example).

The predicate rewrite system used to approximate the Q function in the following experiments is as in Section IV. The predicate rewrite system for the P hypothesis language used in experiments 1, 2, and 4 is as follows.

$$top \rightarrowtail \wedge_3\ top\ top\ top$$
$$top \rightarrowtail extractAction \circ projA \circ top$$
$$top \rightarrowtail extractAction \circ projB \circ top$$
$$top \rightarrowtail extractAction \circ top$$
$$top \rightarrowtail isDeadlocked$$
$$top \rightarrowtail isMisplaced$$
$$top \rightarrowtail isFloor$$
$$top \rightarrowtail isConstructive.$$

In experiment 3, a further three predicate rewrites are required to express the $one\_intention$ policy.

$$top \rightarrowtail achievesSingletonIntention$$
$$top \rightarrowtail notContainIntentionBlock$$
$$top \rightarrowtail aboveIntentionBlock.$$

B. Graphs showing experimental results

Graphs showing the results obtained from the four experiments discussed here are shown at the end of the paper in Figures 3–6.

For each experiment two graphs are plotted, the first depicting cumulative moves versus episodes, the second depicting extra moves versus episodes. *Extra* moves are moves additional to what the policy US predicts for a given episode. US is chosen as a benchmark because it is near optimal for a small number of blocks [11] and it is simple to compute. Note that a consequence of this is that sometimes the number of extra moves plotted in the second type of graph is a small negative number, when the agent solves an episode in fewer moves that US.

C. Experiments 1 and 2 — initialisation

These two experiments consider the initialisation problem for an adaptive agent: how does an agent get started at all? In the blocks world domain the number of possible states rises rapidly when more blocks are considered. Random exploration of such large state spaces to find a goal state is impractical. Some sort of guidance of the agent towards its goal is required while it acquires knowledge of its environment. The symbolic learning system allows the agent's designer to specify an initial policy to guide the agent's initial learning.

Experiment 1 is conducted in a blocks world containing 5 blocks. At each episode the agent is supplied with a new task of achieving a set of 5 on-states. It moves from task to task receiving a reward only for actions which achieve an episode's task.

In the first instance, the agent is supplied with no initial policy and the $selectAction$ function specifies a zero exploration factor. The effect of this is that the agent initially randomly explores the 5-block state space, fully exploiting its (initially empty) current policy that becomes refined through retraining over time.

In the second instance, the agent is supplied with the initial simple-minded policy $policy_{simple}$. The $selectAction$ function again specifies a zero exploration factor. The $updateP$ function delays the collection of training data until after episode 20, and delays the retraining of the initial policy decision tree until after episode 40. The retraining error tolerance is set at 15%.

In both cases the agent eventually converges to a good policy $policy_{GN1}$. This occurs after 52 episodes without guidance and after 71 episodes with guidance, but with fewer overall moves.

Experiment 2 is conducted in a blocks world containing 8 blocks where the task is to achieve a set of 8 on-states. The number of different states in an 8 blocks world is 394,353 compared to 501 for a 5 blocks world (see [11]) and there is a single goal state. Unguided random exploration does not reach the goal state within a practical time, so guided exploration is mandatory. This experiment shows the agent converging to $policy_{GN1}$ after 57 episodes, starting from $policy_{simple}$.

The $selectAction$ function for this experiment specifies an exploration factor of 50% until after episode 20 when it drops to zero. The $updateP$ function delays the collection of training data until after episode 10, and delays the retraining of the initial policy decision tree until after episode 20. The retraining error tolerance is set at 15%.

D. Experiment 3 — adapting to changing tasks

This experiment exhibits the adaptivity of the agent to changing tasks. From time to time an adaptive agent's task might change, either dramatically or incrementally. In the case of a dramatic change in task, the problem of adaptivity reduces to the problem of initialisation (again). In the case of incremental change however, it is desirable that the agent be able to modify its existing behaviour in order to adapt without starting over again.

Experiment 3 is conducted in a 5 blocks world where the initial tasks consist of singleton sets of on-states. That is, the agent is only required to achieve a single on-state. No initial guidance policy is supplied, so the agent initially explores randomly, exploiting its knowledge as it is acquired. The experiment shows that the agent converges to a good policy for achieving singleton on-states $policy_{one\_intention}$ after episode 61. At episode 75, the agent's tasks change and it is subsequently required to achieve sets of 5 on-states at each episode. The order in which it achieves these on-states now becomes important. Rewards are only received for achieving all 5 on-states at the same time. 106 moves are taken to achieve the task in episode 75, during which the agent adopts the more general $policy_{GN1}$ which applies to the new type of tasks and also, with hindsight, to the previous simpler tasks.

The $selectAction$ function for this experiment specifies an exploration factor of zero. The $updateP$ function sets a retraining error tolerance of 5%.

E. Experiment 4 — adapting to a changing environment

This experiment shows the symbolic learning agent adapting to a changing environment. Like the previous case of incremental changes to the agent's tasks, it is also desirable that an agent be able to adapt to changes in its environment without resorting to re-initialisation.

Experiment 4 is conducted in a blocks world initially containing 5 blocks. At some episode, a new block is added to the environment and, at a subsequent episode, yet another block is added. The task of the agent is also incremented in the richer environments from sets of 5 on-states to sets of 6 on-states, and finally to 7 on-states in a 7 blocks world.

The agent is initialised with $policy_{simple}$ and converges to the better policy $policy_{GN1}$ after episode 21. At episode 30, a 6th block is added and the task incremented to 6 on-states. The agent continues to achieve its harder tasks by applying the policy it learned in the simpler world. At episode 37, an inconsistency between the incoming training examples and the current policy is detected to be higher than the error tolerance and a retraining of the current working policy is triggered. This results in a series of suboptimal, but not disastrous, policies until episode 46 at which point they reconverge to $policy_{GN1}$.

At episode 60, a 7th block is added and the task complexity incremented to 7 on-states. $policy_{GN1}$ is maintained until episode 128 when the incoming training data again exceeds the retraining error tolerance triggering a further retraining of the working policy. This results in a suboptimal policy, which is refined over the course of the next few episodes, converging to $policy_{GN1}$ after episode 136.

The $selectAction$ function for this experiment specifies an exploration factor of 50% until after episode 20 when it drops to zero. The $updateP$ function delays the collection of training data until after episode 10, and delays the retraining of the initial policy decision tree until after episode 20. The retraining error tolerance is set at 15%.

VI. Conclusions

We now draw together some general remarks that can be made on the basis of the experiments.

The hypothesis language for the Q function is quite different to the hypothesis language for the policy. The reason is that the Q function encodes the path length to a goal, while the policy distinguishes good and bad actions – they are different functions that therefore require different hypothesis languages.

The policy is primary; the Q function is only used as a 'crutch' to learn a good policy. Furthermore, the Q function does not have to be perfect in order to get a good policy. This is partly because the policy is general, that is, is independent of the size of the state, so small errors in Q can be absorbed by the policy.

Domain knowledge is encoded in the hypothesis languages for the policy and Q functions; specifically it appears in the corresponding predicate rewrite systems. Furthermore, this is a very convenient way of encoding this knowledge, as illustrated by the definitions of the policy and Q functions that are learned; these definitions are compact and comprehensible. The designer or user of the agent can assess the agent's current policy at any time by inspecting the symbolic representation of the policy.

The designer should aim to provide as much domain knowledge as possible to the agent (in the form of the hypothesis languages and the overall knowledge representation chosen) and rely on learning only when really needed.

Discovering a good policy requires not only a good hypothesis language but also good training data. In other words, much attention has to be paid to issues such as adequate exploration of the state space and the size of training windows for the incremental learning algorithm. Too small a window can lose valuable data, while too large a window can restrict the agent's ability to react to a changing environment.

For large search spaces, a good initial policy is highly advantageous in that it can help the agent find rewards more easily (or, even, at all). The symbolic approach helps here as it makes it possible for the designer to easily code up initial policies for this purpose.

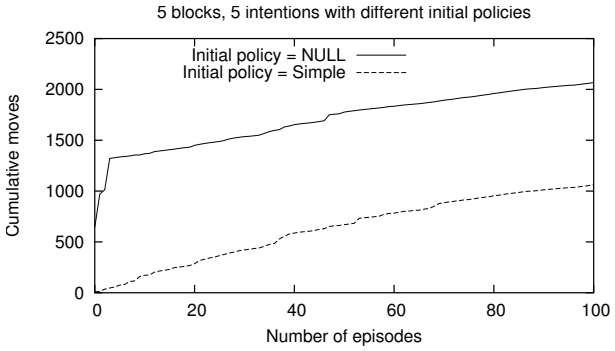
Finally, we remark that methods other than Q learning could be used to learn policies. For example, a model of the environment could be learned and then planning techniques used [3].

ACKNOWLEDGMENT

The authors would like to thank Eric McCreath, Robert Bridle, and Mathi Nagarajan for many helpful discussions concerning the topic of this paper. This research was supported by the Smart Internet Technology Cooperative Research Centre.

REFERENCES

[1] ALKEMY homepage. `http://csl.anu.edu.au/~kee/Alkemy`.
[2] D. Bertsekas and J. Tsitsiklis. *Neuro-dynamic Programming*. Athena Scientific, 1996.
[3] C. Boutilier, T. Dean, and S. Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.
[4] S. Džeroski, L. De Raedt, and H. Blockeel. Relational reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning, ICML'98*, pages 136–143. Morgan Kaufmann, 1998.
[5] S. Džeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001.
[6] M. Georgeff and A. Lansky. Reactive reasoning and planning. In *Proceedings of the Conf. of the American Assoc. of Artificial Intelligence*, pages 677–682, 1987.
[7] J.W. Lloyd. *Logic for Learning*. Cognitive Technologies. Springer, 2003.
[8] K.S. Ng. Incremental induction of Alkemic trees. Technical report, Computer Sciences Laboratory, The Australian National University, 2003.
[9] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.
[10] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, second edition, 2002.
[11] J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125:119–153, 2001.
[12] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
[13] C. Watkins. Q-learning. *Machine Learning*, 8(3):229–256, 1992.



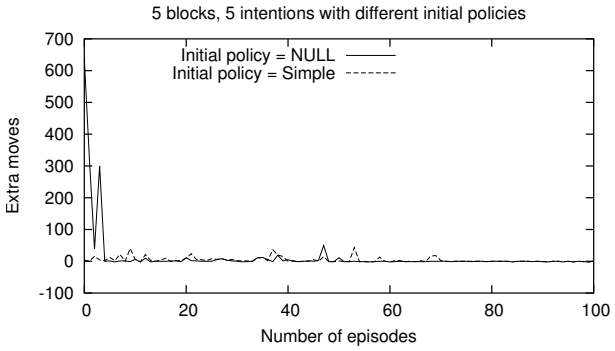
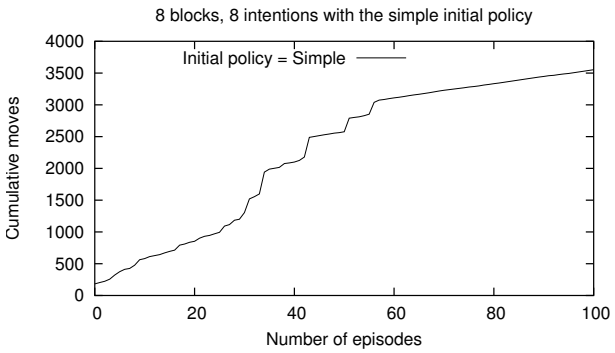


Fig. 3. Experiment 1



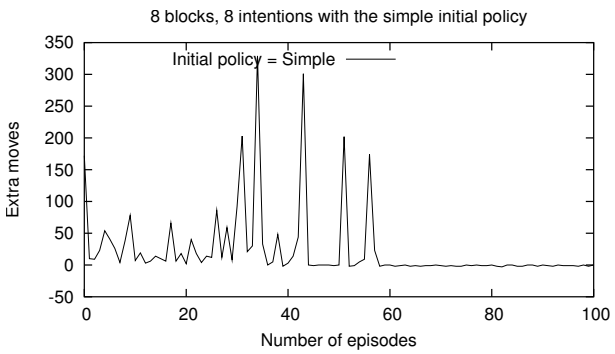
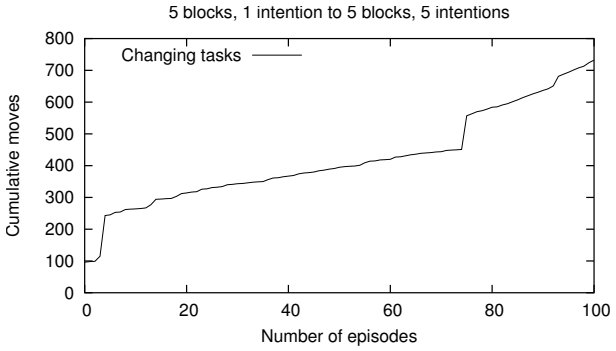


Fig. 4. Experiment 2



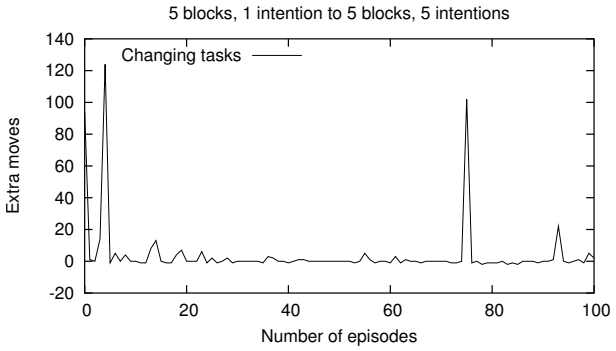
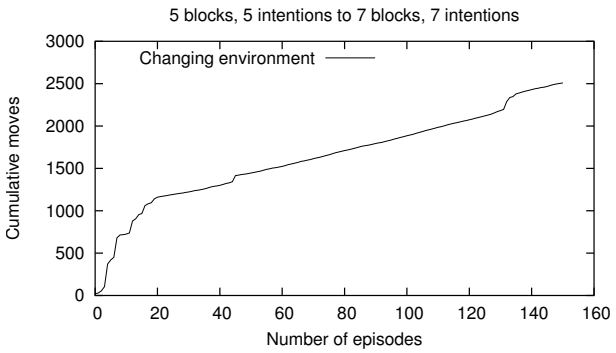


Fig. 5. Experiment 3



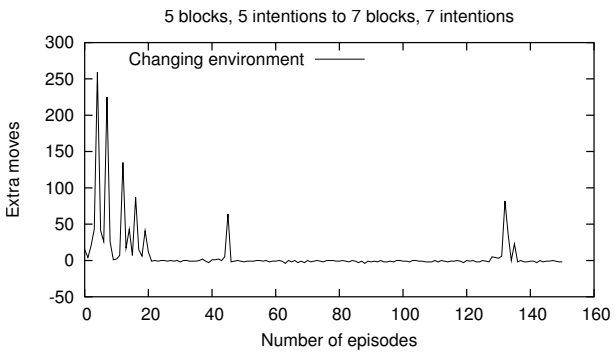


Fig. 6. Experiment 4