Relational Knowledge with Predictive State Representations

David Wingate, Vishal Soni, Britton Wolfe and Satinder Singh
Computer Science and Engineering
University of Michigan, Ann Arbor, MI 48109
{wingated,soniv,bdwolfe,baveja}@umich.edu

Abstract

Most work on Predictive Representations of State (PSRs) has focused on learning and planning in unstructured domains (for example, those represented by flat POMDPs). This paper extends PSRs to represent *relational* knowledge about domains, so that they can use policies that generalize across different tasks, capture knowledge that ignores irrelevant attributes of objects, and represent policies in a way that is independent of the size of the state space. Using a blocks world domain, we show how generalized predictions about the future can compactly capture relations between objects, which in turn can be used to naturally specify relational-style options and policies. Because our representation is expressed solely in terms of actions and observations, it has extensive semantics which are statistics about observable quantities.

1 Introduction

Models of dynamical systems that use predictive representations of state replace the traditional notion of state (which is usually a latent, unobservable quantity) by a set of statistics about the future. A number of results have been obtained about these predictive state representations (or PSRs). In the case of discrete observations, for example, PSRs [Littman et al., 2002] have been shown to be just as accurate, expressive and compact as unstructured partially observable Markov decision processes (POMDPs). In the case of continuous observations and linear dynamics, the Predictive Linear Gaussian model [Rudary et al., 2005] is just as expressive and more compact than the celebrated Kalman filter. Predictive representations have shown to be good bases for generalization [Rafols et al., 2005], and can be learned directly from an agent's experience trajectories by a number of different algorithms [Wiewiora, 2005; McCracken & Bowling, 2006; James & Singh, 2004]. To date, most work on PSRs has focused on state tracking and parameter estimation in flat, unstructured worlds such as general POMDPs and linear dynamical systems.

We extend PSRs in a different direction. Our main contributions are several extensions to the current state-of-the-art of PSRs that allow the sort of generalization that is captured by relational representations. Our extensions allow PSRs to deal with structured relational domains, and entail three natural benefits. First, we are able to express policies for a family of domains in a way that is independent of the size of the state space (for example, in a blocks world domain the description of a policy would be independent of the number of blocks). Second, we can express abstract knowledge by ignoring irrelevant properties of observations (in a blocks world, the color of a block may not impact how it can be manipulated), which allows us to generalize across objects. Third, we can parameterize policies so that they generalize over multiple tasks (in a blocks world, the policy for stacking block a on block b should be similar to stacking block c on block d).

To accomplish this, we broaden the language that PSRs use to represent knowledge. PSRs represent knowledge as action-conditional predictions about the future, which is formally captured by the notion of a *test*. We generalize these tests to include multi-attribute observations, and define two new types of tests, called "set tests" and "indexical tests." We show how policies and options [Precup et al., 1998] can be specified in terms of mappings from these generalized tests to actions, which allows the policies and options to generalize over whatever the tests generalize over. Finally, we show how a nonlinear PSR can be naturally specified with these generalized tests. Our extensions to PSRs maintain a key feature of PSRs, which is that all knowledge, including relational knowledge, has extensive semantics based on statistics about observable quantities. Furthermore, as we will show below, the form of our extensions will allow relational knowledge to be maintained and computed efficiently. In summary, this paper is about advancing the expressiveness of PSRs and extending them to capture relational knowledge. We do not address relational learning or planning.

2 Background and Motivation

The key idea of a PSR is to represent state as a set of *tests*, which represent possible future observations given possible future actions. A PSR maintains a set of *core* tests, the predictions of which are state in any history, and which allow it to make correct predictions about *any* sequence of future events.

In a controlled, discrete, dynamical system (for example, a POMDP), at each time step i, the agent executes an action $a_i \in \mathcal{A}$ and receives an observation $o_i \in \mathcal{O}$ (we occasionally use multi-attribute observations, which are tuples denoted by angle brackets).

A *history* is a sequence of alternating actions and observations $a_1o_1a_2o_2\cdots a_mo_m$ describing an agent's experience through timestep m. An *s-test* (or "sequence test," which we will also just call a "test"), denoted $a^1o^1a^2o^2\cdots a^no^n$, describes a possible sequence of *future* actions and observations (note the distinction between superscripts and subscripts). A test *succeeds* if the observations of the test are obtained, given that the test's actions are taken. A *prediction* for a test $t = a^1o^1a^2o^2\cdots a^no^n$ starting in history h is the probability that t will succeed when its actions are executed immediately following h. Formally, we define the prediction for a test from history h of length m to be $p(t|h) = \Pr(o_{m+1} = o^1, o_{m+2} = o^2, \cdots, o_{m+n} = o^n|h, a_{m+1} = a^1, \cdots, a_{m+n} = a^n)$. For ease of notation, we will use the following shorthand: for a set of tests $T = \{t_1, t_2, \cdots t_n\}$, $p(T|h) = [p(t_1|h), p(t_2|h), \cdots p(t_n|h)]^T$ is a column vector of predictions for the tests in T from a history h.

In a PSR, state is represented by a set of predictions about *core tests*. The predictions of these core tests comprise a sufficient statistic for history, which can be used in two ways. First, they can be used to compute the prediction of an *arbitrary* test [Littman et al., 2002]. Second, they can be used to maintain state, as follows. Given a set of core tests Q, their predictions $p(Q|h)$ (which constitute state), an action a and an observation o, the updated prediction for a core test $q_i \in Q$ is given by

$$p(q_i|hao) = \frac{p(aoq_i|h)}{p(ao|h)} \ .$$

This means that to maintain state, we only need to compute the predictions of the *one step tests* (ao) and the *one-step extensions* (aoq_i) to the core tests as a function of $p(Q|h)$.

2.1 An Example: Blocks World

Blocks world is a structured domain typical of relational settings. Even in this simple world, there are many types of structure that s-test PSRs cannot leverage because the tests they use are not expressive enough.

We define a partially observable blocks world, in which the agent has an eye that can focus on different places (viewing the world from the side). The domain is like a grid world, with discrete positions; for a given position, the observation is either a block, the `Table`, or nothing (represented by ϕ). For most of the paper, we will assume that each block has a unique observation (a number), but we will occasionally consider versions where each block yields a multi-attribute observation. In these cases, the blocks will have a color, a number, and a "marked" bit (explained in Section 5.3).

Actions are `U` (move the eye up), `D` (move the eye down), a_ϕ (stay still) and `find`(a) (saccade to block a). The `move_curr`(b) action moves whatever block the eye is looking at onto block b (both must be clear; b may be the `Table`), and the eye stays where it was. Since we do not specify pre- and postconditions for actions, we assume that if the agent executes an invalid action, it fails and the state is unchanged. The eye can be at most one space above any given stack.

2.2 Structured Questions

In order to exploit relational structure in domains such as the blocks world, we must be able to ask questions about the future that s-tests cannot pose. These questions will motivate our extensions to PSRs, and will allow us to accomplish our goals of generalization across domains, generalization across tasks, and generalization across objects. Here are some examples:

- "What is the probability that if I move my eye down, I will *not* see block b?" This negative question cannot be answered with an s-test because they are defined in terms of things that we *do* see, not things that we do *not* see. However, we can represent "not block b" with the set of all observations except b. This motivates "set tests," which aggregate multiple observations.
- "What is the probability of seeing a red block on top of a blue block?" This type of question motivates an ability to ignore irrelevant attributes of observations, which allows us to deal with objects in terms of their attributes, and not their identities. Again, this motivates set tests.
- "What is the probability of seeing one block, moving up, and seeing a block that has the same (but unspecified) color?" In this case, color is acting as a variable, which must have the same value for both blocks. This question motivates an ability to refer to previous observations, and is the inspiration for "indexical tests."

Asking (and answering) these sorts of questions helps us to capture knowledge of structure in a domain, but requires generalizing the language of s-tests. One possibility is to allow tests to contain arbitrary questions, represented by arbitrary functions of observations and action. However, the answers (or predictions) of such arbitrary tests may not be efficiently computable. We seek a simpler way: we will show that our generalized tests provide the needed capability for capturing relational knowledge, and are computationally efficient.

3 Generalizing the Language of Tests

This section discusses two generalizations of standard s-tests. We will first define set tests, in which we replace a single possible observation with a set of acceptable observations, and will then discuss indexical and complemented indexical tests.

3.1 Set Tests

Standard s-tests can only ask about one observation at each timestep, but to ask negative tests and to ignore irrelevant attributes we want to ask about arbitrary sets of observations. We accomplish this with *set tests*.

Suppose we have two tests t_1 and t_2, both of length m, that differ in a only a single observation. We could ask, "What is the probability of t_1 or t_2?" Because only one of the tests could succeed from a given history, the answer is $p(t_1|h) + p(t_2|h)$. Set tests generalize this idea by replacing a single observation with an arbitrary set of observations. Define a set of observations $O^k \subseteq \mathcal{O}$, and let $\sigma^k_j \in O^k$ be the j-th member of O^k. Define a test t which replaces the k-th observation o^k with the set O^k: $t = a^1o^1\cdots a^kO^k\cdots a^no^n$. This test can be viewed as a set of $|O^k|$ tests (individually denoted t_j), where t_j replaces O^k with σ_j^k. Then $p(t|h) = \sum_{j=1}^{|O^k|} p(t_j|h)$.

There is considerable flexibility in defining these sets of observations. To ask about red objects, we construct a set of all observations with the attribute "red." We are also not limited to replacing a single observation with a set; any (or all) observations could be replaced. If we replace multiple observations, the resulting test includes all combinations of the elements of those sets. For example, if $O^1 = \{\sigma_1^1, \sigma_2^1\}$ and $O^2 = \{\sigma_1^2, \sigma_2^2\}$, the test $t = a^1 O^1 a^2 O^2$ "expands" to include four primitive tests: $a^1\sigma_1^1 a^2 \sigma_1^2$, $a^1\sigma_1^1 a^2\sigma_2^2$, $a^1\sigma_2^1 a^2\sigma_1^2$, and $a^1\sigma_2^1 a^2\sigma_2^2$, and the prediction of t is the sum of their predictions.

Because the systems we consider have a finite set of observations, the complement of an observation is well defined. To ask negative tests, we let $\neg o^k = \mathcal{O} \setminus \{o^k\}$, which is the set of all observations except o^k.

3.2 Indexical Tests

With a set test we can ask, "What is the probability of seeing a red block on top of another red block?", but a different kind of test is needed to ask "What is the probability of seeing two blocks on top of each other which have the same (but unspecified) color?"

We call these tests *indexical tests*. In an indexical test, we replace a single observation o^k with a variable X^k, and then we replace some later observation o^f with the *same* variable, as in $t = a^1 o^1 \cdots a^k X^k \cdots a^f X^k \cdots a^n o^n$. Of course, multiple observations can be replaced, and indexical variables can be reused many times.

By default, the variable X^k will "match" any observation from $\mathcal{O}$, but we can restrict it to match a subset of observations. In either case, we define $\mathtt{dom}(X^k) \subseteq \mathcal{O}$ to be the set of acceptable observations for X^k. For multi-attribute observations, X^k is a *vector* of variables $[X_1^k, \ldots, X_m^k]^T$, any of which we can restrict. This allows us to test whether observations match on some attributes of interest, while ignoring the other attributes. For example, $t = a^1\langle X_1^1, X_2^1, X_3^1\rangle a^2 \langle X_1^1, Y_2^2, Y_3^2\rangle$, with no restriction on the domains, will succeed if the first attribute is the same in both time steps. We do not allow indexical variables to bind across tests.

Let us contrast indexical tests and set tests. Earlier, we showed a set test $t_O = a^1 O^1 a^2 O^2$ that expanded to include four tests. Suppose we let $\mathtt{dom}(X^1) = O^1$. Then, the indexical test $t_X = a^1 X^1 a^2 X^1$ only includes two primitive tests ($a^1\sigma_1^1 a^2\sigma_1^1$ and $a^1\sigma_2^1 a^2\sigma_2^1$), and the prediction of t_X is the sum of their predictions.

It is natural to define the indexical analog of a complemented set test. A *complemented indexical test* asks the following kind of question: "What is the probability that if I take some action, see some observation, and take another action, I will see a *different* observation?" To do this, we allow $\neg X^k$ in a test, in addition to X^k.

3.3 Linear PSRs and Efficient Tests

One of the primary motivations for restricting the class of generalized tests we consider is computational. There are special efficiencies for our generalized tests in the case of linear PSRs, which allow the predictions of the generalized tests to be computed in time that is independent of the "complexity" of the test. Before explaining further, we will first review linear PSRs.

In a linear PSR, for every s-test t, there is a weight vector $m_t \in \mathbb{R}^{|Q|}$ independent of history h such that the prediction $p(t|h) = m_t^T p(Q|h)$ for all h. This means that updating the prediction of a single core test $q_i \in Q$ can be done efficiently in closed-form. From history h, after taking action a and seeing observation o:

$$p(q_i|hao) = \frac{p(aoq_i|h)}{p(ao|h)} = \frac{m_{aoq_i}^T p(Q|h)}{m_{ao}^T p(Q|h)}$$

We only need to know m_{ao}, which are the weights for the *one-step tests*, and the m_{aoq_i}, which are the weights for the *one-step extensions*.

An important computational aspect of set tests is the fact that their predictions can also be computed with *linear* operations. In particular, for any set test t, there is a weight vector m_t such that $p(t|h) = m_t^T p(Q|h)$. Let $t = a^1 o^1 \cdots a^k O^k \cdots a^n o^n$, where O^k is a set of observations. Then it is easy to show that

$$m_t^T = m_{a^n o^n}^T M_{a^{n-1}o^{n-1}} \cdots \left(\sum_{\sigma^k \in O^k} M_{a^k \sigma^k}\right) \cdots M_{a^1 o^1}$$

where the i-th row of $M_{ao} \in \mathbb{R}^{|Q|\times|Q|}$ is $m_{aoq_i}^T$. In the case that multiple observations are replaced with sets, we compute the weight vector by replacing the corresponding M_{ao} with the appropriate sum.

The prediction for an indexical test is also linear. Let $t = a^1 o^1 \cdots a^k X^k \cdots a^f X^k \cdots a^n o^n$. Then:

$$m_t^T = \sum_{\sigma^k \in \mathtt{dom}(X^k)} m_{a^n o^n}^T \cdots M_{a^k \sigma^k} \cdots M_{a^f \sigma^k} \cdots M_{a^1 o^1}.$$

In the case of multiple, reused or complemented indexicals, the weight vector equation generalizes in the obvious way: the outer sum is over all valid joint settings of the indexical variables.

Because the weight vectors for set and indexical tests are independent of history, they can be computed once and reused over and over again. Thus, after a one-time cost of computing the weights, the complexity of using both set and indexical tests is independent of the length of the test as well as the number of observations being aggregated (equivalently, the number of s-tests being aggregated). The complexity does, however, depend on the number of core tests.

4 Specifying Options and Policies with Generalized Tests

We will now show how generalized tests can be used to specify policies and options. Two of the three criteria for success outlined in the introduction involve the use of compact, descriptive policies: first, the complexity of a policy should be independent of the size of our domain, and second, it should generalize over multiple (but similar) tasks. We show that by specifying options and policies in terms of generalized tests, both automatically generalize in the same way their constituent tests do.

4.1 Specifying Policies with Tests

Formally, a policy is a mapping from states to actions. States in a PSR are represented as the predictions of core tests $p(Q|h)$. This state will always be a vector in $\mathbb{R}^{|Q|}$, so a policy must map $\mathbb{R}^{|Q|}$ to $\mathcal{A}$.

Instead of mapping $p(Q|h)$ directly to actions, we first use the state, $p(Q|h)$, to compute the predictions of a set of additional generalized tests, and then map the predictions of those additional tests to actions. There are many possible ways to do this. One way is to create a list of expressions involving tests, associate an action with each expression, and execute the first action whose expression is true. For example:

$$\begin{aligned} p(\mathtt{U}, \phi|h) = 1 &\quad\rightarrow\quad \mathtt{move\_curr(Table)} \\ p(a_\phi, \phi|h) = 1 &\quad\rightarrow\quad \mathtt{D} \\ \mathtt{default} &\quad\rightarrow\quad \mathtt{U} \end{aligned} \tag{1}$$

This policy moves the eye to the top of the stack, and then pops blocks off the stack until it is empty. It operates independently of the number of blocks in the domain, as well as the number of blocks in the stack. To avoid notational clutter in deterministic domains, we will usually omit the $p(\cdot|h) = 1$ associated with each test.

4.2 Specifying Options with Tests

Options are temporally extended actions. They could represent, for example, "walk to the doorway," or "wait until the light turns green." Formally, an option is specified by a set of initiation states (in which the option is available), by a termination function (which specifies the probability that the option will terminate at each timestep, and which may be a function of state), and by a policy over states.

We specify the initiation states, the termination function, and the option policy in terms of tests. An option policy is specified in the same way as a global policy. The initiation states and termination function are also specified in the same way, but each expression has an associated value in $\{0, 1\}$ or $[0, 1]$, respectively, rather than an associated action. For example, using the policy in Eq. (1), allowing initiation in any state, and using

$$\begin{aligned} p(a_\phi, \langle O^1, O^2, O^3\rangle|h) \wedge p(\mathtt{U}, \phi|h) = 1 &\quad\rightarrow\quad 1.0 \\ \mathtt{default} &\quad\rightarrow\quad 0.0 \end{aligned}$$

as our termination condition, then we have an option that pops blocks off the stack until the top block matches the criteria specified by $\langle O^1, O^2, O^3\rangle$. Here we define $p(t_1|h) \wedge p(t_2|h) \equiv p(t_1|h)p(t_2|h)$.

5 Connecting to Relational Knowledge Representations

We have completed the development of our generalized tests, and will next demonstrate how they can be applied with a detailed example. First, we will define predictive equivalents to traditional relational predicates and will illustrate their use in several options and policies. The first goal we consider is $\mathtt{on}(a, b)$, but we will also develop a policy for the more complicated goal of $\mathtt{on}(\langle X, *, *\rangle, \langle X, *, *\rangle)$, which is satisfied when two blocks of the same color are stacked (regardless of what color they are). Finally, we will show how predictive predicates can create a nonlinear PSR to track the state of the world.

Predicate	Test	Predicate	Test
$\mathtt{on}(a,b)$	$= \mathtt{find}(a), a, \mathtt{D}, b$	$\mathtt{eye\_on}(a)$	$= a_\phi, a$
$\mathtt{clear}(a)$	$= \mathtt{find}(a), a, \mathtt{U}, \phi$	$\mathtt{eye\_above}(a)$	$= \mathtt{D}, a$
$\neg\mathtt{on}(a,b)$	$= \mathtt{find}(a), a, \mathtt{D}, \neg b$	$\mathtt{eye\_below}(a)$	$= \mathtt{U}, a$
$\neg\mathtt{clear}(a)$	$= \mathtt{find}(a), a, \mathtt{U}, \neg\phi$		

Figure 1: Table of predictive predicates used.

5.1 Predictive Predicates and Core Tests

A traditional relational representation of the blocks world defines several predicates such as $\mathtt{clear}(a)$ and $\mathtt{on}(a, b)$. Equivalent constructs, which we will call *predictive predicates*, can be defined predictively with generalized tests. Predictive predicates form building blocks for defining policies and options whose semantics are entirely based on statistics about future observations.

For example, the test $t_{\mathtt{on}(a,b)} \equiv a^1 = \mathtt{find}(a), o^1 = a, a^2 = \mathtt{D}, o^2 = b$, can be used to define a predictive predicate $\mathtt{on}(a, b)$ that is is true whenever $p(t_{\mathtt{on}(a,b)}|h) = 1$. This test finds block a, moves the eye down and checks to see if it sees block b. Similarly, the test $t_{\mathtt{clear}(a)} \equiv a^1 = \mathtt{find}(a), o^1 = a, a^2 = \mathtt{U}, o^2 = \phi$ can be used to define a predictive predicate $\mathtt{clear}(a)$ that is true in a history h whenever $p(t_{\mathtt{clear}(a)}|h) = 1$. For ease of exposition, we will omit the implied tests associated with each of these predictive predicates, and just write $\mathtt{on}(a, b)$ instead of $p(t_{\mathtt{on}(a,b)}|h)$; we will also omit the implied action and observation labels, writing $\mathtt{on}(a, b) = \mathtt{find}(a), a, \mathtt{D}, b$. To track the position of the eye, we define three predictive predicates: $\mathtt{eye\_on}(a) = a_\phi, a$, $\mathtt{eye\_above}(a) = \mathtt{D}, a$, and $\mathtt{eye\_below}(a) = \mathtt{U}, a$. We can also discuss the complement of a predictive predicate in terms of a complemented set test. For example, $\neg\mathtt{on}(a, b) = \mathtt{find}(a), a, \mathtt{D}, \neg b$.

To emphasize that these predictive predicates are actually tests, Fig. 1 summarizes the predicates and their corresponding tests.

5.2 Options and Policies

Given those predictive predicates, we can now begin to define options and policies that accomplish abstract actions in our domain. For instance, the following option, $\mathtt{pop\_curr}$, moves to the table the block at the top of the current stack. This option works independently of the number of blocks in the stack, which is important to deal with stacks independently of the size of the state space:

Option:	$\mathtt{pop\_curr}$	
Available:	$\neg\mathtt{eye\_on}(\phi)$	
Termination:	$\mathtt{eye\_on}(\phi)$	
Policy:	$\mathtt{eye\_below}(\phi)$	$\rightarrow$ $\mathtt{move\_curr(Table)}$
	$\neg\mathtt{eye\_below}(\phi)$	$\rightarrow$ $\mathtt{U}$

Given the $\mathtt{pop\_curr}$ option and the rest of our predictive predicates, we can define an option to satisfy $\mathtt{on}(a, b)$. The idea of this policy is to find block a, clear all of the blocks off the top of it, then find block b and clear it, then move block a onto block b:

Option:	$\texttt{on}(a,b)$	
Available:	(all)	
Termination:	$\texttt{on}(a,b)$	
Policy:		
$\texttt{eye\_on}(a) \wedge \texttt{clear}(a) \wedge \texttt{clear}(b)$	$\rightarrow$	$\texttt{move\_curr}(b)$
$\neg\texttt{eye\_on}(a) \wedge \texttt{clear}(a) \wedge \texttt{clear}(b)$	$\rightarrow$	$\texttt{find}(a)$
$\texttt{eye\_on}(b) \wedge \texttt{clear}(a) \wedge \neg\texttt{clear}(b)$	$\rightarrow$	$\texttt{pop\_curr}$
$\neg\texttt{eye\_on}(b) \wedge \texttt{clear}(a) \wedge \neg\texttt{clear}(b)$	$\rightarrow$	$\texttt{find}(b)$
$\texttt{eye\_on}(a) \wedge \neg\texttt{clear}(a)$	$\rightarrow$	$\texttt{pop\_curr}$
$\neg\texttt{eye\_on}(a) \wedge \neg\texttt{clear}(a)$	$\rightarrow$	$\texttt{find}(a)$

There are a few points worth noting about this policy. The policy is compact and independent of the size of the state space (that is, it works for any number of blocks). The existence of such a policy is not a surprise. Indeed, the policy is a straightforward implementation of a standard relational policy in PSR terms. What is interesting is that PSRs allow one to write such a policy in purely observable quantities. Note that the policy is only good for stacking a on b; later, we will parameterize this policy to generalize across similar tasks.

5.3 More Sophisticated Goal States

So far, we have only used set tests. To showcase the power of indexical tests, we will now develop policies for more sophisticated goal states. The first is $\texttt{on}(\langle X,*,*\rangle, \langle X,*,*\rangle)$, which is satisfied when any two blocks of the same color are stacked. We now have a multi-attribute observations, which are color, number, and a "marked" bit. The agent is allowed to $\texttt{mark}$ blocks as a form of visual memory; only one block can be marked at a time.

We also need to make a few intuitive redefinitions and clarifications to actions and predicates. Basically, we allow everything to take an observation set as an argument. For example, $\texttt{find}(O^i)$ now moves the eye randomly to a block that satisfies O^i (each satisfactory block has a nonzero probability of being reached). Predictive predicates are similar: we define $\texttt{eye\_on}(O^i) = a_\phi, O^i$ $\texttt{eye\_above}(O^i) = \texttt{D}, O^i$, and $\texttt{eye\_below}(O^i) = \texttt{U}, O^i$. The on predictive predicate is a little different because the generalized test $\texttt{find}(O^i), O^i, \texttt{D}, O^j$ is no longer deterministic. We define on to be true when $p(\texttt{find}(O^i), O^i, \texttt{D}, O^j|h) > 0.0$. In other words, it is an existential relational query that has nonzero probability when there is some object satisfying O^i on top of some object satisfying O^j. The $\texttt{clear}$ predictive predicate is similar: $\texttt{clear}(O^i) = (\texttt{find}(O^i), O^i, \texttt{U}, \phi) > 0.0$.

To start constructing our policy, we will define an indexical test which is true whenever the marked block has the same color as the block the eye is looking at:

$$\texttt{same\_as\_marked} = a_\phi, \langle X,*,\neg\texttt{mark}\rangle, \texttt{find}(\langle *,*,\texttt{mark}\rangle), \langle X,*,\texttt{mark}\rangle$$

This lets us define $\texttt{pop\_until\_same\_as\_marked}$, which is an option to pop blocks off the current stack until the top block has the same color as the marked block:

Option:	$\texttt{pop\_until\_same\_as\_marked}$	
Available:	(all)	
Termination:	$\texttt{same\_as\_marked} \wedge \texttt{eye\_below}(\phi)$	
Policy:		
$\texttt{eye\_below}(\phi)$	$\rightarrow$	$\texttt{move\_curr(Table)}$
$\texttt{eye\_on}(\phi)$	$\rightarrow$	$\texttt{D}$
$\texttt{default}$	$\rightarrow$	$\texttt{U}$

We define another option, $\texttt{pop\_until}(\langle O^1, O^2, O^3\rangle)$, which is defined by Eq. (1) and the termination condition in Section 4.2. Together, these two options help us create a policy for $\texttt{on}(\langle X,*,*\rangle, \langle X,*,*\rangle)$. The idea is to find a block, mark it, and then clear all blocks off the top of it. Then, find another block with the same color, clear it, then put them on top of each other:

Option:	$\texttt{on}(\langle X,*,*\rangle, \langle X,*,*\rangle)$	
Available:	(all)	
Termination:	$\texttt{on}(\langle X,*,*\rangle, \langle X,*,*\rangle)$	
Policy:		
$\texttt{clear}(\langle *,*,\texttt{mark}\rangle) \wedge$ $\texttt{same\_as\_marked} \wedge$ $\texttt{eye\_below}(\phi)$	$\rightarrow$	$\texttt{move\_curr}(\langle *,*,\texttt{mark}\rangle)$
$\texttt{clear}(\langle *,*,\texttt{mark}\rangle) \wedge$ $\texttt{same\_as\_marked}$	$\rightarrow$	$\texttt{pop\_until\_same\_as\_marked}$
$\texttt{clear}(\langle *,*,\texttt{mark}\rangle)$	$\rightarrow$	$\texttt{find}(\langle *,*,\neg\texttt{mark}\rangle)$
$\texttt{eye\_on}(\langle *,*,\texttt{mark}\rangle)$	$\rightarrow$	$\texttt{pop\_until}(\langle *,*,\texttt{mark}\rangle)$
$\texttt{eye\_on}(\texttt{Table} \cup \phi)$	$\rightarrow$	$\texttt{find}(\langle *,*,*\rangle)$
$\texttt{default}$	$\rightarrow$	$\texttt{mark}$

We could easily extend this policy to solve for a goal like $\texttt{on}(\langle \texttt{red},*,*\rangle, \langle \texttt{red},*,*\rangle)$ by replacing the $\texttt{find}$ action with an option that finds a $\texttt{red}$ block.

5.4 Parameterized Policies and Options

So far, our policies and options generalize across domains and across objects, but do not generalize across tasks. For example, the policy for achieving $\texttt{on}(a,b)$ does not generalize to stacking other blocks on each other. However, since all of our tests and predicates are propositional in nature, we could parameterize the tests and actions and options in the obvious way: simply replace constant observations with variables. So, $\texttt{on}(a,b)$ might become $\texttt{on}(A,B) = \texttt{find}(A), A, \texttt{D}, B$ for variables A and B. As long as actions can be similarly parameterized, the agent can bind these variables to specific blocks and obtain predictions and policies on-demand.

5.5 A Nonlinear, Relational PSR

So far, we have tacitly assumed the existence of an underlying PSR capable of predicting any test. We will now show how such a PSR can be constructed using predictive predicates. Two things are needed for a PSR: a set of core tests and a state update mechanism.

We represent our core tests as predictive predicates. If there are k blocks, one set of core tests is k^2 predicates, representing each possible $\mathtt{on}(a,b)$ combination, plus $O(k)$ additional predicates tracking the eye and marker. Practically, this is an overkill: since our domain is deterministic, we can compress to just $O(k)$ predicates by remembering one $\mathtt{on}(a,b)$ predicate for each a (this complexity is analogous to the fact that in a traditional relational system, the size of the knowledge base grows linearly with the number of blocks in the domain, although the policy description does not).

To update the state of the system, we use a rule-based system. We associate a rule with every action, which describes how to update the prediction of each core test. An example of such a rule is shown below:

```
Action: move_curr(d)
  clear(d) ← 0;   eye_on(φ) ← 1
  ∀X: if eye_on(X)          | ∀X,Y: if eye_on(X) ∧ on(X,Y)
      eye_on(X) ← 0         |     on(X,Y) ← 0
      on(X,d) ← 1           |     clear(Y) ← 1
```

This rule is run whenever the action $\mathtt{move\_curr}(d)$ is executed. It updates both the core tests representing what the eye sees, as well as all of the core tests representing the state of the blocks (the PSR is nonlinear because of the nonlinear nature of these rules).

6 Related Work

This paper focuses on extending PSRs to allow generalized tests, which can be used to build relational knowledge into PSR models, as well as build generalizing policies and options. We do not yet address learning and planning with this new representation (however, see [Dzeroski et al., 2001] for an example of existing work on learning with logical relational representations). Furthermore, the representations developed here are essentially propositional, and are at present unrelated to efforts to combine first-order relational knowledge with probability [Milch et al., 2005; Richardson & Domingos, 2006]. Temporal difference networks [Sutton & Tanner, 2005] are perhaps the model closest in spirit to ours. While it is theoretically possible for a TD-net to compute predictions for our generalized tests, they were not specifically designed for such a task and would require numerous modifications. In contrast, our generalized tests function very naturally in a PSR setting: their predictions are linearly computable, they arise from a natural generalization of s-tests, and they can succinctly represent relational knowledge.

7 Conclusions and Future Research

We have taken the first steps in extending PSRs from their existing limitation to flat, unstructured domains into domains that have relational structure. We introduced set tests and indexical tests, which allow an agent to ask (and answer) questions that s-tests and e-tests cannot represent. We showed that policies and options can be defined in terms of these generalized tests. We have illustrated these policies and options, as well as how to build a PSR model that captures relational knowledge, in a prototypical blocks world. We also discussed how all of these tests, policies, and options may be parameterized to further generalize.

We connected our generalized tests to traditional relational representations, and showed how standard relational predicates can be defined in predictive terms. We used these predictive predicates to translate the traditional blocks world domain into a PSR model, with a final form that is closely connected to a relational representation. Our resulting relational PSR model satisfies our three basic criteria for success: policy succinctness, policy generalization, and attribute irrelevance.

As immediate future work, we are addressing the problem of learning PSR models that use these generalized tests and then planning with them in relational domains.

Acknowledgments

Special thanks to Matt Rudary for generous help and insightful comments. David Wingate is supported under a National Science Foundation Graduate Research Fellowship. Britton Wolfe, Vishal Soni and Satinder Singh are supported by NSF grant IIS-0413004, NSF grant CCF-0432027, and by a grant from DARPA's IPTO program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or DARPA.

References

Dzeroski, S., Raedt, L. D., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*, 7–52.

James, M. R., & Singh, S. (2004). Learning and discovery of predictive state representations in dynamical systems with reset. *ICML* (pp. 417–424).

Littman, M. L., Sutton, R. S., & Singh, S. (2002). Predictive representations of state. *NIPS* (pp. 1555–1561).

McCracken, P., & Bowling, M. (2006). Online discovery and learning of predictive state representations. *NIPS* (pp. 875–882).

Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D. L., & Kolobov, A. (2005). Blog: Probabilistic models with unknown objects. *IJCAI* (pp. 1352–1359).

Precup, D., Sutton, R. S., & Singh, S. P. (1998). Theoretical results on reinforcement learning with temporally abstract options. *ECML* (pp. 382–393).

Rafols, E. J., Ring, M. B., Sutton, R. S., & Tanner, B. (2005). Using predictive representations to improve generalization in reinforcement learning. *IJCAI* (pp. 835–840).

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine Learning*, *62*, 107–136.

Rudary, M. R., Singh, S., & Wingate, D. (2005). Predictive linear-Gaussian models of stochastic dynamical systems. *UAI* (pp. 501–508).

Sutton, R. S., & Tanner, B. (2005). Temporal-difference networks. *NIPS* (pp. 1377–1384).

Wiewiora, E. (2005). Learning predictive representations from a history. *ICML* (pp. 964–971).