THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Learning Knowledge-Level Domain Dynamics

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Author final version (often known as postprint)

Published In:
Proceedings of the ICAPS 2013 Workshop on Planning and Learning

OPEN ACCESS

Learning knowledge-level domain dynamics

Kira Mourão
School of Informatics
University of Edinburgh
Edinburgh, EH8 9AB, UK
kmourao@inf.ed.ac.uk

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh, EH8 9AB, UK
rpetrick@inf.ed.ac.uk

Abstract

The ability to learn relational action models from noisy, incomplete observations is essential to support planning and decision-making in real-world environments. While some methods exist to learn models of STRIPS domains in this setting, these approaches do not support learning of actions at the knowledge level. In contrast, planning at the knowledge level has been explored and in some domains can be more successful than planning at the world level. In this paper we therefore present a method to learn knowledge-level action models. We decompose the learning problem into multiple classification problems, generalising previous decompositional approaches by using a graphical deictic representation. We also develop a similarity measure based on deictic reference which generalises previous STRIPS-based approaches to similarity comparisons of world states. Experiments in a real robot domain demonstrate our approach is effective.

Introduction

The related problems of planning and learning domain dynamics in domains with incomplete knowledge and uncertainty are both challenging. The planning problem has been tackled using the possible worlds paradigm (Weld *et al.*, 1998; Bonet and Geffner, 2000; Bertoli *et al.*, 2001), where the planner reasons about actions across all possible worlds in which the agent might be operating given its current knowledge. An alternative is to use a knowledge-level representation that describes the agent's knowledge without enumerating possible worlds. One such approach is to restrict the agent's knowledge to simple relational and functional properties using *knowledge fluents*, and then plan with these structures either directly (Petrick and Bacchus, 2002, 2004) or indirectly through compilation techniques (Palacios and Geffner, 2009), in an attempt to build plans more efficiently. However, while a few approaches have tackled learning domain dynamics with incomplete knowledge (Amir and Chang, 2008; Zhuo *et al.*, 2010; Mourão *et al.*, 2012), none have considered learning knowledge-level actions, such as would be required by a planner operating directly at that level.

In this paper we present a method for learning action rules in knowledge domains. We consider the problem of acquiring domain models from the raw experiences of an agent exploring the world, where the agent's observations are incomplete, and observations and actions are subject to noise.

The domains we consider are based on relational STRIPS domains (Fikes and Nilsson, 1971) but also include functions, run-time variables and knowledge fluents.

We tackle the problem of learning action models from noisy and incomplete observations by decomposing the problem into multiple classification problems, similar to the work of Halbritter and Geibel (2007) and Mourão *et al.* (2009; 2010; 2012). Our approach generalises these earlier approaches by using a decomposition based on a deictic representation. We represent world states as graphs and develop a similarity measure, also based on deictic reference, to perform similarity comparisons between states. The features used to measure similarity are closely related to the rules underlying the true action models. We reuse the rule extraction method of Mourão *et al.* (2012) to derive planning operators from classifiers trained using our new representation.

We test our approach in a real robot domain. The robot bartender (Petrick and Foster, 2013) serves drinks to customers by generating plans based on input from its vision and dialogue processing systems. State observations derived from these systems can be incomplete or noisy, due to sensing errors. Therefore states are modelled at the knowledge level, with functions and run-time variables used to capture customer requests. Our experiments show that the domain models we learn for the robot bartender perform as well as a "gold-standard" hand-written domain model used to generate the robot's plans.

The Learning Problem

A *domain* $\mathcal{D}$ is defined as a tuple $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{F}, \mathcal{A} \rangle$, where $\mathcal{O}$ is a finite set of world objects, $\mathcal{P}$ is a finite set of predicate (relation) symbols, $\mathcal{F}$ is a finite set of function symbols, and $\mathcal{A}$ is a finite set of actions. Each predicate, function, and action also has an associated arity. A *fluent expression* of arity n is a statement of the form:

(i) $p(c_1, c_2, ..., c_n)$, where $p \in \mathcal{P}$, and each $c_i \in \mathcal{O}$, or

(ii) $f(c_1, c_2, ..., c_n) = c_{n+1}$, where $f \in \mathcal{F}$, and each $c_i \in \mathcal{O}$.

A *real-world state* is any set of positive or negative fluent expressions, and $\mathcal{S}$ is the set of all possible states. State observations may be incomplete, so we assume an open world where unobserved fluents are deemed to be unknown. At the world level, for any state $s \in \mathcal{S}$, fluent ϕ is true at s iff $\phi \in s$, and false at s iff $\neg\phi \in s$. A fluent and its negation cannot both be in s. If $\phi \notin s$ and $\neg\phi \notin s$ then ϕ is unobserved.

At the knowledge level we transform state observations of the real world into *knowledge states*: statements about the agent's knowledge of the world. A *knowledge fluent* $K\phi$ denotes whether a real-world fluent ϕ is known to be true in the world ($K\phi$), false in the world ($K\neg\phi$) or unknown ($\neg K\phi$ and $\neg K\neg\phi$). Therefore at the knowledge level the closed world assumption can be reinstated and whenever both $K\phi \notin s$ and $K\neg\phi \notin s$, we know that $\neg K\phi \in s$ and $\neg K\neg\phi \in s$. Additionally we introduce the operator K_v which indicates whether the value of a function $f(c_1, c_2, \ldots, c_n)$ is known ($K_v(f(c_1, c_2, \ldots, c_n))$) or unknown ($\neg K_v(f(c_1, c_2, \ldots, c_n))$), regardless of the actual value. Thus $(\exists d \in \mathcal{O})K(f(c_1, \ldots, c_n) = d) \equiv K_v(f(c_1, \ldots, c_n))$. All states at the knowledge level are written entirely in terms of these knowledge fluents.

Each action $a \in \mathcal{A}$ is defined by a set of *preconditions*, Pre_a, and a set of *effects*, Eff_a. Pre_a can be any set of knowledge fluent expressions. We consider two different kinds of action effects. First, we allow STRIPS-like effects, where each $e \in Eff_a$ has the form *add*(ϕ), or *del*(ϕ), and ϕ is any knowledge fluent expression. Second, we permit *conditional effects* of the form $C_e \Rightarrow add(\phi)$ or $C_e \Rightarrow del(\phi)$. Here, C_e is any set of knowledge fluent expressions, and is referred to as the *secondary preconditions* of effect e. Action preconditions and effects can also be parameterised. An action with all of its parameters replaced with objects from $\mathcal{O}$ is an *action instance*.

In contrast to STRIPS domains, which assume that objects mentioned in the preconditions or the effects must be listed in the action parameters (the *STRIPS scope assumption* (SSA)), we make the more general *deictic scope assumption* that objects mentioned in the preconditions or effects are either action parameters or are directly or indirectly related to the action parameters, i.e., they have a deictic term (see Deictic Reference section).

We restrict previous domain knowledge to the assumption of a weak domain model where the agent knows how to identify objects, has acquired predicates to describe object attributes and relations, and knows what types of actions it may perform, but not the appropriate contexts for the actions, or their effects. Experience in the world is then developed by observing changes to object attributes and relations when "motor-babbling" with primitive actions.

The task of the learning mechanism is to learn the preconditions and effects Pre_a and Eff_a for each $a \in \mathcal{A}$, from data generated by an agent performing a sequence of randomly selected actions in the world and observing the resulting states. The sequence of states and action instances is denoted by $s_0, a_1, s_1, ..., a_n, s_n$ where $s_i \in \mathcal{S}$ and a_i is an instance of some $a \in \mathcal{A}$. Our data consists of *observations* of the sequence of states and action instances $s'_0, a_1, s'_1, ..., a_n, s'_n$, where state observations may be noisy (some $\phi \in s_i$ may be observed as $K\neg\phi \in s'_i$) or incomplete (some $\phi \in s_i$ are not in s'_i). Action failures are allowed: the agent may attempt to perform actions whose preconditions are unsatisfied. In these cases the world state does not change, but the observed state may still be noisy or incomplete. To make accurate predictions in domains where action failures are permitted, the learning mechanism must learn

both preconditions and effects of actions.

Consider, for example, the dishwasher domain (shown in Figure 1), a domain where an agent can load and unload a dishwasher, switch it on, and check the status of the dishwasher. In our examples we use a PDDL-like syntax to represent knowledge fluents and states. For a state where the agent knows the dishwasher contains some dirty dishes, the real world state could be:

```
(AND (status=dirty) (¬in washer dish1) (¬in washer dish2)
    (in washer dish3) (isdirty dish1) (¬isdirty dish2)
    (isdirty dish3) (in washer dish4) (isdirty dish4)).
```

From this the agent might observe the knowledge state:

```
(AND Kᵥ(status) K(status=dirty) K(¬in washer dish1)
    K(in washer dish3) K(isdirty dish1) K(isdirty dish3)
    K(¬in washer dish2) K(¬isdirty dish2)).
```

A sequence of knowledge states and actions could be:

```
s₀:(AND Kᵥ(status) K(status=dirty) K(¬in washer dish1)
    K(in washer dish3) K(isdirty dish1) K(isdirty dish3)
    K(¬in washer dish2) K(¬isdirty dish2))
a₁:(load washer dish1)
s₁:(AND Kᵥ(status) K(status=dirty) K(in washer dish1)
    K(in washer dish3) K(isdirty dish1) K(isdirty dish3)
    K(¬in washer dish2) K(¬isdirty dish2))
a₂:(switchon)
s₂:(AND K(in washer dish1) K(in washer dish3)
    K(¬in washer dish2) K(¬isdirty dish2))
a₃:(checkstatus)
s₃:(AND K(in washer dish1) K(in washer dish3)
    K(¬in washer dish2) K(¬isdirty dish2)
    Kᵥ(status) K(status=clean)).
```

Taking a sequence of such inputs, we learn action descriptions for each action in the domain, such as in Figure 1.

Related Work

Knowledge-level reasoning is not a new idea (Newell, 1982), and the use of knowledge fluents like $K\phi$ and $K\neg\phi$ has been explored as a means of restricting the syntactic form of knowledge assertions in exchange for more tractable reasoning, e.g., by avoiding the drawbacks of possible-worlds models (Demolombe and Pozos Parra, 2000; Soutchanski, 2001; Petrick and Levesque, 2002). Planners like PKS (Petrick and Bacchus, 2002, 2004) attempt to work directly with knowledge-level models, similar to those of knowledge fluents, while approaches like (Palacios and Geffner, 2009) compile traditional open world planning problems into a classical closed-world form, in the process automatically generating knowledge fluents.

Only a few approaches to learning action models are capable of learning under either partial observability (Amir and Chang, 2008; Yang *et al.*, 2007; Zhuo *et al.*, 2010), noise in any form (Pasula *et al.*, 2007; Rodrigues *et al.*, 2010), or both (Halbritter and Geibel, 2007; Mourão *et al.*, 2010). Some rely on prior knowledge of the action model, such as using known successful plans (Yang *et al.*, 2007; Zhuo *et al.*, 2010), or excluding action failures (Amir and Chang, 2008). None explicitly support functions or knowledge fluents.

While the representation used in our previous work (Mourão *et al.*, 2012) does not support functions or the K_v

```
(define (domain dishwasher)
  (:predicates (in ?washer ?dish) (isdirty ?dish))
  (:functions (status ?washer) = ?washstatus)
  (:constants clean dirty)

(:action checkstatus
  :parameters (?washer)
  :precondition ()
  :effect (K_v(status(?washer))))

(:action switchon
  :parameters (?washer)
  :precondition ()
  :effect (¬K_v(status(?washer))))

(:action unload
  :parameters (?washer ?dish)
  :precondition (and K_v(status(?washer)) K(status(?washer)=clean) K(in ?washer ?dish))
  :effect (and K(¬in ?washer ?dish) K(¬isdirty ?dish)))

(:action load
  :parameters (?washer ?dish)
  :precondition (K¬(in ?washer ?dish) )
  :effect (and K(in ?washer ?dish)
          (when (K(isdirty ?dish)) (K_v(status(?washer)) K(status(?washer)=dirty)))
          (when (and ¬K(isdirty ?dish) K_v(status(?washer)) K(status(?washer)=clean)) (¬K_v(status(?washer))))))))
```


Figure 1: A description of the dishwasher domain (left), and (right) a graphical representation of state s_0 when combined with the `load` action. The node representing the result of the `status(?washer)` function is labelled $status_v$.

operator, it could support knowledge fluents of the form $K\phi$. In this earlier work, each fluent ϕ was assigned one of the values 1, -1 or $*$ which correspond to the $K\phi$, $K\neg\phi$ and $\neg K\phi/\neg K\neg\phi$ defined earlier. However, the learning method depended on the SSA to generate vector representations of states. With the introduction of functions the SSA no longer applies and the vector representation can no longer be used.

Our new approach depends upon coding world states (and correspondingly, preconditions and effects) in terms of deictic reference (Agre and Chapman, 1987). A deictic representation maintains pointers to objects of interest in the world, with objects coded relative to the agent or current action. Previous work in learning action models has also used deictic reference (Benson, 1996; Pasula *et al.*, 2007) because there are benefits in doing so: it reduces the size of the state representation, by limiting the observations to a small number of objects, and also permits generalisation across different instances of the same action, as the observations are described in terms of the action and the agent instead of specific objects.

Method outline

Our approach to learning knowledge-level action models is based on the work of Mourão *et al.* (2012), but differs significantly in terms of the representation used and in the details of the learning process. Real-world states are observed by an agent as a knowledge state where each fluent $\phi(\neg\phi)$ is observed as $K\phi(K\neg\phi)$ and when $Kf(c_1,\ldots,c_n) = c_{n+1}$, also $K_v(f(c_1,\ldots,c_n))$. We represent these observations as graphs where objects, *known* fluents and actions are nodes in the graph, and edges link fluents to their arguments. The prediction problem is then to determine which nodes in a

graph change as the result of an action. Our strategy is to decompose the prediction problem into many smaller classification problems, where each classifier predicts change to a single fluent of the overall state, given an input situation and an action. After training the classifiers we derive planning operators from the learnt parameters, using the same process described by Mourão *et al.* (2012).

Central to the classification process is a measure of similarity between states. Commonly, similarity comparisons between graphs are performed using graph kernels which implicitly map into another feature space; here we define an explicit mapping of state graphs into a feature space, where the mapping is calculated via a simple relabelling scheme.

The remainder of this paper is structured as follows. We define deictic reference and show how it is used to create the graphical representation of world states. Then we explain how we calculate a similarity measure for two states based on deictic reference. The structure and operation of the classification learning model is described, followed by an explanation of how rules are extracted from the classifiers. Finally, we give some experimental results and discuss conclusions and future work.

Deictic reference

Deictic reference underlies a number of aspects of the learning process. The structure of the state observation graphs is determined by the deictic terms of the objects in the state. In turn, this means that the feature space mapping relies on deictic reference to map objects with the same roles in an action to the same points in the feature space.

In the deictic representation we use, we code objects with respect to the action. Every action parameter is referred to

by its own unique deictic term, corresponding to its position in the parameter list. Constant values are also considered to have their own deictic terms. Deictic terms referring to other objects are their definitions in terms of their relations with the action parameters and other objects.

Thus, similar to Pasula *et al.* (2007), a deictic term is a variable V_i and a constraint ρ_i where ρ_i is a set of literals defining V_i in terms of the arguments of the current action and any previously defined V_j ($j < i$). Then an object has a deictic term if it is an argument of the current action, or it is related directly, or indirectly via other objects, to the arguments of the action. For functions, every argument must already have a deictic term in order for the function result to have a deictic term.

Additionally, we add the constraint that for an object to have a deictic term, it must be linked by a positive fluent to either an action parameter, or another object which has a deictic term (the *positive link assumption*). This additional restriction accounts for the open world representation now in place (at the world level), avoiding deictic terms of the form "the-object-not-under-the-object-I-am-picking-up-and-not-on-the-floor", which will not usually be unique and seem counter-intuitive. Apart from the action parameters, any object in a state may be referred to by several deictic terms, and (unlike Pasula *et al.* (2007)) any deictic term may refer to several objects in a state.

We say that an object has an n-th order deictic term when n is the minimum number of relations relating the object to an action parameter. Thus the parameters of the action have zero-order deictic terms, while objects related to the action parameters have first-order deictic terms.

For example, in the dishwasher domain (Figure 1), if the action were (load washer dish1) in state s_0, then action parameters washer and dish1 would have deictic terms arg_1 and arg_2, indicating their positions in the load argument list. Relative to the (load washer dish1) action, dish2 is referred to by deictic terms $x : \neg in(washer, dish2)$ and $x : \neg in(washer, dish2) \land \neg isdirty(x)$, but not $x : \neg isdirty(x)$ alone. The dish2 node is labelled [dish2] to indicate that it represents all objects with the same deictic terms as dish2.

State representation

We represent a knowledge state by a graph, where objects (as deictic terms), known fluents, and the current action are represented by nodes in the graph. Edges link fluents (or the current action) and their arguments, and are labelled with the argument position.

Both predicates and functions are represented by nodes and are only present in the graph if known. However, for functions additionally the result of a function f is represented by a special node f_v, which denotes the deictic term defined by the function. The actual value of the function is linked to f_v by an equality node. Thus, for example, $K(f(c_1, c_2) = c_3)$ would be represented as in Figure 2.

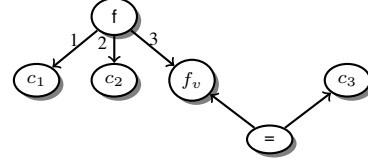The size of the graph is limited by restricting the deictic terms to zero- or first-order terms only.[1] Using only zero-

---

[1]Higher order terms are possible but are left to future work.

Figure 2: Representation of $K(f(c_1, c_2) = c_3)$. c_1,c_2 and c_3 are represented by nodes labelled with their deictic terms (here we assume they are constants). The function node f has edges to nodes c_1 and c_2, indicating they are parameters, and also an edge connecting to the result node f_v. f_v and c_3 are linked by an equals node, indicating that the value of $f(c_1, c_2)$ is c_3.

order terms would be equivalent to working with a STRIPS representation, as we would only consider parameters of the action during learning. Here, we require first-order deictic terms to represent functions, as the result of a function will not usually be an action parameter. Figure 1 shows a graph encoding the state s_0 in the context of the (load washer dish1) action, after converting the objects to deictic terms.

Calculating changes

Our classification model operates by taking a knowledge state (as a graph) as input, and predicting which knowledge fluents will change. Each training example must therefore consist of a prior state, an action, and the changes resulting from performing the action on the state.

We denote changes by creating a *change graph*, created by annotating the prior state graph with additional *marker* nodes (similar to Halbritter and Geibel (2007)). Marker nodes have an edge linking to the fluent node which changed. Given a prior and successor state, a marker node M_ϕ is added to the change graph for every fluent ϕ which changes real-world value between the states. A marker node $M_{K\phi}$ is added for every fluent which changes knowledge state between the states. During training, each classifier will learn to predict the presence or absence of a single marker node in the graph (i.e. whether the associated fluent changes).

It is straightforward to determine the marker nodes to add to the change graph, given prior and successor state graphs. For any fluent ϕ in the prior state, if $\neg\phi$ is in the successor state, we add M_ϕ. If neither ϕ nor $\neg\phi$ are present in the successor state we add $M_{K\phi}$. Similarly, any fluent present in the successor state but not the prior state is added to the change graph, along with $M_{K\phi}$. For example, for the load action in Figure 1, the changes to the state would be indicated by a node $M_=$ linked to the (status$_v$ = clean) node and a node M_{in} linked to the (¬in arg$_1$ arg$_2$) node.

Crucially, because the successor state immediately follows the prior state, matching fluents can be determined by matching the actual objects which were arguments of the fluents. In general such matching is not possible between states. We return to this point when describing the structure of the learning model.
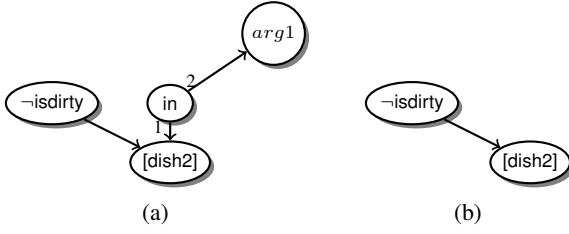
Figure 3: Valid (a) and invalid (b) subgraphs of the state graph in Figure 1.

Comparing states using deictic reference

The classification process requires a measure of similarity between states. In classification problems, graphical inputs are usually mapped either implicitly — via graph kernels — or explicitly into a feature space where the inner product provides a similarity score.

A feature space where the features are all possible conjunctions of fluents would seem to be ideal for learning action preconditions which are arbitrary conjunctions of fluents. However, similarity calculations in this space are unlikely to be tractable as it is closely related to the subgraph kernel (mapping graphs to the space of all possible subgraphs), known to be NP-hard (Gärtner *et al.*, 2003), and contains the feature space of the DNF kernel (Sadohara, 2001; Khardon and Servedio, 2005), which cannot be used by a perceptron to PAC-learn DNF (Khardon *et al.*, 2005).

Following Mourão *et al.* (2012) we therefore work with the space of all possible conjunctions of fluents of length $\leq k$ for some fixed k. The space is further restricted so that in every conjunction, every object must have a valid deictic term depending only on fluents in the conjunction. This restriction avoids learning meaningless preconditions where variables in the preconditions are undefined e.g., action $a(x, y)$ with precondition $p(z)$. Also, it forces the similarity comparison to account for the roles of objects (as defined by their deictic terms) by mapping objects in different states, but with similar deictic terms, to similar sets of features.

We define an explicit mapping into this space, creating a (sparse) feature vector. Each element of the vector corresponds to a conjunction of up to k fluents present in the state graph, subject to the restriction that every object has a valid deictic term depending only on fluents in the conjunction. E.g. considering subgraphs of the dishwasher state shown in Figure 1, Figure 3a would be valid but not Figure 3b. The value of each element in the vector is the number of occurrences of the corresponding subgraph in the state graph.

The feature vector can be constructed via a labelling scheme similar to the process used in some graph kernel calculations (Shervashidze *et al.*, 2011). First we label object nodes with either their position in the action parameter list, or their type if they are not listed in the action parameters. Next we identify the set of *core* fluents, whose arguments are contained within the set of action parameters. By definition, every argument of a core fluent has a deictic term, and so any conjunction of core fluents will be valid.

For each conjunction C of i core fluents ($1 \leq i \leq k$), we identify the set of *supported* fluents, whose arguments are also arguments of either the action or a fluent in C. For example, in Figure 3a, `in` is a core fluent and `isdirty` is a supported fluent. Every argument of a supported fluent will have a deictic term depending only on fluents in C. Now we create all possible conjunctions of supported fluents of size $k - i$ or fewer, and combine each with C in turn to give C'.

We convert each fluent in C' to a string encoding the fluent, the argument positions and their ordering. E.g. `(in arg1 dish)` could convert to "in1(arg1)2(dish)". (Note that here "dish" is a type.) Next we sort the fluent strings and concatenate them to give a unique string representing C'. This string is looked up in a lookup table mapping strings to feature vector locations. If the string is not found in the lookup table, we add a new entry with value 1 to the feature vector and a matching entry in the lookup table. Otherwise we increment the existing entry in the feature vector.

Structure of the learning model

Using the state graphs defined above, the structure of the learning model can be defined. Given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, the model predicts the successor state s'. Equivalently, the set of fluents which change between s and s' — the deltas — can be predicted. Our strategy is to use multiple classifiers where each classifier predicts change to one or a small set of fluents of the overall state, given an input situation and an action.

Such a structure requires a classifier for each possible fluent node in any state graph. Then given a state graph, we predict the effect of an action by predicting whether each fluent node in the graph changes or not. The conjunction of all the predicted changes is the predicted effect of the action. For example, in Figure 1, consider the following fluents:

1. `(¬in arg₁ arg₂)`
2. `(¬in arg₁ [dish2])`
   where $[dish2] = \{x : \neg in(arg_1, x) \wedge \neg isdirty(x)\}$
3. `(in arg₁ [dish4])`
   where $[dish4] = \{x : in(arg_1, x) \wedge \neg isdirty(x)\}$
4. `(¬in arg₁ [dish5])`
   where $[dish5] = \{x : \neg in(arg_1, x)\}$

Fluents (1) and (2), present in the graph, and (3), not present, but possible, would each have their own classifier. Additionally we must consider fluents with more general deictic terms, such as (4), which includes both (1) and (2). The classifier associated with (4) predicts whether fluent `(in arg₁ x)` changes for *any* x not in arg_1, whereas the classifiers associated with (1) and (2) predict whether `(in arg₁ x)` changes for x which is the second argument of the `load` action (1), or for x which is not in arg_1 and not dirty (2). However, although there are many possible fluent nodes, in practice most of the associated classifiers are not instantiated by our algorithm, resulting in a default prediction of no change for the corresponding fluents.

Our training algorithm therefore has two tasks. First, it manages sets of classifiers, in terms of deciding which classifier to train on which data, and when to instantiate new classifiers. Second, it trains the classifiers. Likewise, at prediction our algorithm must select which classifiers to use, and then generate a prediction from them.

As in the work of Mourão *et al.* (2012), we will use voted perceptron classifiers (Freund and Schapire, 1999), since they are known to be robust to noise and efficient to train. We use the standard procedures for training of, and prediction from, individual classifiers. In our algorithm descriptions below, $train(c, x, y)$ denotes updating classifier c with training example (x, y), and $predict(c, x)$ returns classifier c's prediction of the class of example x. We now describe how classifiers are managed during training and prediction.

Initialisation

The algorithm is provided with the set of action labels $\mathcal{A}$, the set of predicates $\mathcal{P}$, the set of functions $\mathcal{F}$, and the number and types of their arguments. In the following description we treat any function $f(c_1, \ldots, c_n) = c_{n+1}$ as two predicates: $f'(c_1, \ldots, c_n, f_v)$ and $equals(f_v, c_{n+1})$, corresponding to the graph structure defined earlier, and contained in an extended set $\mathcal{P}'$. The learning algorithm maintains a set of classifiers $C_{a,p}$ for each action a and predicate p. Initially each $C_{a,p}$ is empty and is populated as training examples are seen by the algorithm. Every member of $C_{a,p}$ will be a classifier $c_{\overline{m}}$ associated with a different tuple of deictic terms $\overline{m}$ which are valid arguments of p. For example, in our dishwasher domain, one of the sets of classifiers would be $C_{(load,in)}$: the set of classifiers which predict changes to the in predicate when the load action is performed. A member of $C_{(load,in)}$ could be $c_{(arg_1, \{x:in(arg_1,x) \land \neg isdirty(x)\})}$.

Training

Each training example consists of a state description x_i, an action a_i, and a successor state x_i'. Both state descriptions are converted into state graphs and a change graph δ_i, based on the action a_i as previously described. The marker nodes from the change graphs will provide target values.

The training process is outlined in Algorithm 1. In the main loop we identify all the fluent nodes $p(\overline{m})$ in a training example x ($fluentNodes(x)$) and determine whether each fluent changed in the example, by checking whether the node has a marker node in the change graph δ ($isFluentInDelta$). If the fluent changed, the target value y is set to 1, otherwise it is set to 0. Then $updateClassifiers$ is called for each fluent node.

In $updateClassifiers$, classifiers which match $p(\overline{m})$ are trained, and new classifiers may be instantiated if necessary. Recall that in principle there is one classifier for every possible fluent, each initially predicting no change to the fluent. 'No-change' classifiers are not actually instantiated since no prediction function is needed. During training, $updateClassifiers$ must decide which classifiers to update, i.e., first, whether to instantiate a classifier, and second, which classifier(s) to train. There is also a secondary goal of minimising the number of instantiated classifiers to keep the calculation tractable.

Thus given any $p(\overline{m})$ we first seek classifiers which predict for $p(\overline{m})$ and then update them with the training example (x, y). A classifier predicts for $p(\overline{m})$ if it is labelled with $p(\overline{m})$ (an exact match) or labelled with $p(\overline{m}')$ where $\overline{m}'$ is equal to or more general than $\overline{m}$ (a subset match). For example, if $q(\{x : a(x) \land b(x)\})$ is a unary predicate then

---

Algorithm 1 Training

Require: training egs $(x_1, a_1, \delta_1), ..., (x_n, a_n, \delta_n) \in X$
Ensure: trained classifiers
1: $C_{a,p} := \varnothing \;\; \forall a \in \mathcal{A}, \forall p \in \mathcal{P}$
2: **for all** $(x, a, \delta) \in X$ **do**
3:     **for all** $p(\overline{m}) \in fluentNodes(x)$ **do**
4:       $y := isFluentInDelta(p(\overline{m}), \delta)$
5:       $C_{a,p} := updateClassifiers(x, y, \overline{m}, C_{a,p})$

function $updateClassifiers$(state graph x, target y, deictic terms $\overline{m}$, set of classifiers C)
1: $exactMatch := false$; $intersectMatches := \varnothing$
2: **for all** $c \in C$ **do**
3:     **if** $subsetMatch(c, \overline{m})$ **then**
4:       **call** $train(c, x, y)$
5:       **call** $updateReliability(c)$
6:     **if** $exactMatch(c, \overline{m})$ **then**
7:       $exactMatch := true$
8:     **else if** $intersectMatch(c, \overline{m})$ **then**
9:       $intersectMatches := intersectMatches \cup \{c\}$
10: **if** $(y \neq 0) \land (exactMatch = false)$ **then**
11:     $C := C \cup createClassifiers(x, intersectMatches, \overline{m})$
12: **return** C

$q(\{x : a(x)\})$ is more general, and so whenever the former changes, so will the latter. Thus whenever we update $c_{q(\{x:a(x) \land b(x)\})}$ we must also update $c_{q(\{x:a(x)\})}$. Formally, we define that if classifier c predicts change for $p(\overline{n})$:

- $exactMatch(c, \overline{m})$ when $\overline{n} = \overline{m}$;
- $subsetMatch(c, \overline{m})$ if the i-th term in $\overline{n}$ is a subset of the i-th term in $\overline{m}$ $\forall i$;

Any classifier $c \in C_{a,p}$ for which $subsetMatch(c, \overline{m})$ holds is trained on the training example (x, y), and a measure of its reliability updated (see below).

Next we consider whether any classifiers should be instantiated. There are two cases where instantiation is required. If there was no exactly matching classifier for $p(\overline{m})$ *and* in our training example $p(\overline{m})$ changed, then $c_{p(\overline{m})}$ should be instantiated. If $p(\overline{m})$ did not change then the original 'no-change' classifier is still correct. Additionally, the deictic terms seen in training examples may be more specific than the underlying rules. For example if a and b are deictic terms we may only ever see changes to $p(a, arg1)$ or $p(b, arg1)$ but the true change could be to $p(a \cap b, arg1)$. To predict change to the correct set of fluents we therefore need to consider more general deictic terms, and so whenever a new classifier is instantiated, classifiers for tuples of more general deictic terms are also instantiated. However, it is undesirable to add a classifier for every possible tuple, so only those supported by the data are added. These are cases where the deictic terms of $p(\overline{m})$ intersect with deictic terms of $p(\overline{n})$ already seen in the data. Such $p(\overline{n})$ can be found by considering the terms of previously instantiated classifiers.

Formally, if classifier c predicts change for $p(\overline{n})$: $intersectMatch(c, \overline{m})$ if the i-th term in $\overline{n}$ intersects the i-th term in $\overline{m}$ $\forall i$. A tally is kept of exact matches and intersect matches for $p(\overline{m})$, and if $c_{p(\overline{m})}$ in instantiated, so are classifiers for all the intersecting cases ($createClassifiers$).

Algorithm 2 Prediction

Require: Unlabelled instance (x, a), model parameters $C_{a,p}$
Ensure: Prediction δ
1: $\delta = \varnothing$
2: **for all** $p(\overline{m}) \in fluentNodes(x)$ **do**
3:    **if** $getPrediction(C_{a,p}, x, \overline{m}) = 1$ **then**
4:       $\delta = \delta \cup \{p(\overline{m})\}$

function $getPrediction$(set of classifiers C, state graph x, deictic terms $\overline{m}$)
1: $r := 0, y := 0$
2: **for all** $c \in C$ **do**
3:    **if** $subsetMatch(c, \overline{m})$ and $r < getReliability(c)$ **then**
4:       $y := predict(c, x)$
5:       $r := getReliability(c)$
6: **return** y

Reliability and Prediction

The algorithm maintains a reliability score for each classifier ($updateReliability$), used during prediction to select the best classifier. The reliability of a classifier is calculated as the fraction of predictions made which were correct during training. We also maintain the *null reliability*, the reliability which would have been achieved if this classifier had always predicted no change. The null reliability score is thus the fraction of training examples where there was no change. In noisy situations, the null reliability may be higher than the classifier reliability, indicating that many training examples were noisy. In this case, predicting no change gives better results than using the classifier's predictions (on the training set). During prediction, $getReliability$ returns either the classifier reliability or the null reliability, whichever is higher. If the null reliability is higher $predict$ will always predict no change, instead of the classifier's prediction. (Additionally, although not used here, low reliability classifiers can be deleted if the number of classifiers grows too large.)

At prediction, given a test example x, each fluent node $p(\overline{m})$ of x is considered in turn and a search for matching classifiers is performed. If no classifiers are found then the model predicts no change for the fluent $p(\overline{m})$. If exactly one classifier is found then its prediction is used, and if there are multiple matching classifiers, the classifier with the highest reliability score is used.

Learning planning operators

Once the classifiers are trained, planning operators can be derived using the approach of Mourão *et al.* (2012). First, rules are extracted from individual classifiers. Since each voted perceptron classifier predicts change to a single fluent, this results in a set of candidate preconditions for each candidate effect. Second, the candidate preconditions and effects are combined via a heuristic merging process to produce planning operators. These steps are outlined below.

Algorithm 3 Rule extraction

Require: Positive support vectors SV^+
Ensure: Rules $R = \{rule_v : v \in SV^+\}$
1: **for** $v \in SV^+$ **do**
2:    $child := v$
3:    **while** $child$ only covers +ve training examples **do**
4:       $parent := child$
5:       **for** each fluent node in $parent$ **do**
6:          flip node to its negation and calculate weight
7:       $child :=$ child whose parents have least weight difference
8:    $rule_v := parent$

Extracting rules from individual classifiers

Extracting rules from individual classifiers in the graphical case is a straightforward reapplication of the approach used for STRIPS vectors (Mourão *et al.*, 2012). A key point is that the decision function of the voted perceptron is a function of the set of support vectors identified during learning, where the set of support vectors is some subset of the set of training examples.[2]

Rules are extracted from a voted perceptron with kernel K and support vectors $SV = SV^+ \cup SV^-$, where SV^+ (SV^-) is the set of support vectors whose *predicted* values are 1 (-1). Value 1 means the corresponding fluent changes, and -1 means there is no change. The positive support vectors are each instances of some rule learnt by the perceptron, and so are used to "seed" the search for rules. The extraction process aims to identify and remove all irrelevant nodes in each support vector, using the voted perceptron's prediction calculation to determine which nodes to remove.

We define the *weight* of any possible state graph $\mathbf{x}$ to be the value calculated by the voted perceptron's prediction calculation before thresholding. The basic intuition behind the rule extraction process is that more discriminative features will contribute more to the weight of an example. Thus the rule extraction process operates by taking each positive support vector and repeatedly deleting the fluent node which contributes least to the weight until some stopping criterion is satisfied. This leaves the most discriminative features underlying the example, which can be used to form a precondition. This algorithm is detailed in Algorithm 3.

Combining rules into planning operators

Finally we combine the rule fragments ((precondition,effect) pairs) resulting from the rule extraction process into planning operators. For each action the process derives a rule (g_{rule}, e_{rule}) from the set of rules $R = \{(g_1, e_1), \ldots, (g_r, e_r)\}$ produced by rule extraction, ordered by decreasing weight. The process first initialises g_{rule} to the highest weighted precondition in R and sets $e_{rule} = \varnothing$. The rule is then refined by combining it with each of the remaining per-fluent rules in turn, in order of highest weight.

Combining rules involves merging the graphs encoding the preconditions, as well as the markers encoding the effects, into a new candidate rule. After merging, a simplifica-

---

[2]Note that support vectors are therefore state graphs.

tion step removes unnecessary fluents in the preconditions and effects by testing the coverage and weight of the candidate rule without each new fluent. Then the new rule is accepted if its F-score on the training set is within some tolerance of the F-score of the previous rule on the training set. Lastly the rule is translated into PDDL or some variant.

Experiments

We evaluate our approach by learning planning operators in a real robot domain, whose underlying model is defined at the knowledge level. We compare the F-scores for predictions made by both the learnt planning operators and underlying classification model with predictions made by the "gold-standard" domain description: the original specification of the behaviour of the robot.

The data used for training and testing was generated from logs of the JAMES robot bartender system, recorded during a drink ordering scenario in which human subjects were asked to order drinks from the robot. State descriptions were generated by the system's state manager, based on real-world sensor data (vision and automatic speech recognition), interleaved with the names of planned actions generated for the goal of serving all agents. In total, 93 interactions were recorded for 31 human users. Each interaction involves approximately 5-10 robot actions.

The robot bartender domain description is at the knowledge-level, and several actions require functions in their definitions. One action is of particular interest: `ask-drink`, where the robot asks a human customer for their order. If successful, `ask-drink` has the effect that the robot now knows the value of the customer's requests (K_v(request ?x)). Although `ask-drink` will also result in the robot knowing the actual drink requested (e.g. K(request(?x) = water)) this is only useful at run-time, whereas K_v(request ?x) is needed at plan-time. Furthermore, because `ask-drink` involves accurately interpreting the user's chosen drink, it is particularly prone to failure. Therefore it is of additional interest to investigate how well this action is learnt.

Results

A ten-fold cross-validation procedure was used to test the performance of the learning model, and was repeated across different numbers of training examples to assess how many examples would be needed to learn an adequate model. The performance was measured by considering the fluents which the model predicted would change versus the fluents which did change, and calculating the F-score, the harmonic mean of precision and recall (true positives/predicted changes and true positives/actual changes, respectively).

The results were compared to the predictions made by the gold-standard model. In Figure 4 we show F-scores for action predictions made by the classifiers; by rules derived from the classifiers; and by the gold-standard model on data from the robot experiment. As can be seen in the graph, the rules extracted from the classifiers perform similarly to making predictions directly with the classifiers, but with the added benefit of providing action descriptions which can
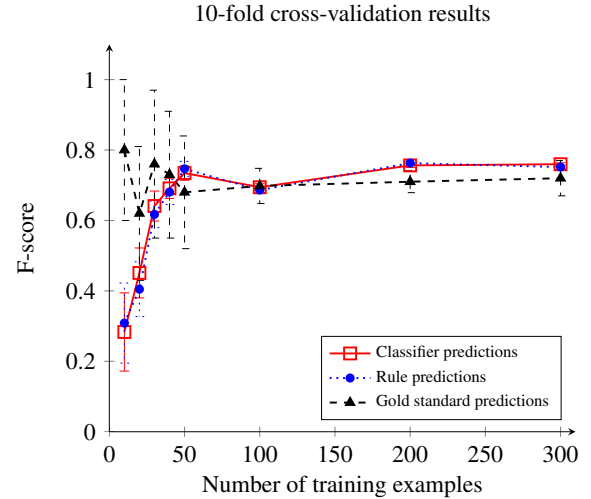

Figure 4: Results from the robot experiment: Mean F-scores from ten-fold cross-validation for predictions from the classifiers, extracted rules and gold-standard action descriptions.

be used for planning. The F-scores for the classifiers and extracted rules are not significantly different from the F-score of the gold standard rules (noise in the domain means that even the gold-standard rules cannot always predict the changes which will or will not occur).

An example of an action description learnt for `ask-drink` with 200 training examples is given below. Fluents marked in italic do not exist in the gold standard domain description. Some fluents are also missing, all relating to preconditions involving other agents which we currently do not represent. However, the crucial K_v(request ?x) effect is learnt.

```
(:action ASK-DRINK
 :parameters (?x)
 :precondition (AND K(transHistory RobotAckAttention ?x)
          K(¬transHistory AgentOrdered ?x)
          ¬Kᵥ(request ?x) K(closeToBar ?x) K(faceseen ?x))
 :effect (AND (Kᵥ(request ?x)
            K(transHistory AgentOrdered ?x))))
```

Conclusions and Future Work

Our results show that we can learn knowledge-level planning operators in a noisy robot domain. The approach we use depends on decomposing the learning problem into many small classification problems, using the deictic scope assumption to constrain the problem. Deictic reference also plays an important role in defining the representation for functions and in the similarity calculations made by the classifiers. In future work we plan to test our approach in other real or simulated knowledge-level domains. Another step will be to use the learnt planning operators in an automated knowledge-level planning system such as PKS (Petrick and Bacchus, 2002, 2004).

References

Agre, P. E. and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *AAAI*, pages 268–272.

Amir, E. and Chang, A. (2008). Learning partially observable deterministic action models. *JAIR*, **33**, 349–402.

Benson, S. S. (1996). *Learning Action Models for Reactive Autonomous Agents*. Ph.D. thesis, Stanford University.

Bertoli, P., Cimatti, A., Roveri, M., and Traverso, P. (2001). Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proc. of IJCAI 2001*, pages 473–478.

Bonet, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *Proc. of AIPS 2000*, pages 52–61.

Demolombe, R. and Pozos Parra, M. P. (2000). A simple and tractable extension of situation calculus to epistemic logic. In *Proc. of ISMIS 2000*, pages 515–524.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, **2**, 189–208.

Freund, Y. and Schapire, R. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, **37**, 277–96.

Gärtner, T., Flach, P., and Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives. In *Proc. of COLT 2003*, pages 129–143.

Halbritter, F. and Geibel, P. (2007). Learning models of relational MDPs using graph kernels. In *Proc. of MICAI 2007*, pages 409–419.

Khardon, R. and Servedio, R. A. (2005). Maximum margin algorithms with Boolean kernels. *JMLR*, **6**, 1405–1429.

Khardon, R., Roth, D., and Servedio, R. A. (2005). Efficiency versus convergence of Boolean kernels for on-line learning algorithms. *JAIR*, **24**, 341–356.

Mourão, K., Petrick, R. P. A., and Steedman, M. (2009). Learning action effects in partially observable domains (1). In *Proc. of ICAPS 2009 Workshop on Planning and Learning*, pages 15–22.

Mourão, K., Petrick, R. P. A., and Steedman, M. (2010). Learning action effects in partially observable domains (2). In *Proc. of ECAI 2010*, pages 973–974.

Mourão, K., Zettlemoyer, L., Petrick, R. P. A., and Steedman, M. (2012). Learning STRIPS operators from noisy and incomplete observations. In *Proc. of UAI 2012*, pages 614–623.

Newell, A. (1982). The knowledge level. *Artif. Intell.*, **18**(1), 87–127.

Palacios, H. and Geffner, H. (2009). Compiling uncertainty away in conformant planning problems with bounded width. *JAIR*, **35**(1), 623–675.

Pasula, H., Zettlemoyer, L. S., and Kaelbling, L. P. (2007). Learning symbolic models of stochastic domains. *JAIR*, **29**, 309–352.

Petrick, R. P. A. and Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In *Proc. of AIPS 2002*, pages 212–221.

Petrick, R. P. A. and Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proc. of ICAPS 2004*, pages 2–11.

Petrick, R. P. A. and Foster, M. E. (2013). Planning for social interaction in a robot bartender domain. In *Proc. of ICAPS 2013, Special Track on Novel Applications*. To appear.

Petrick, R. P. A. and Levesque, H. (2002). Knowledge equivalence in combined action theories. In *Proc. of KR 2002*, pages 303–314.

Rodrigues, C., Gérard, P., and Rouveirol, C. (2010). Incremental learning of relational action models in noisy environments. In *Proc. of ILP 2010*, pages 206–213.

Sadohara, K. (2001). Learning of Boolean functions using support vector machines. In *Proc. of ALT*, pages 106–118.

Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., and Borgwardt, K. M. (2011). Weisfeiler-Lehman graph kernels. *JMLR*, **12**, 2539–2561.

Soutchanski, M. (2001). A correspondence between two different solutions to the projection task with sensing. In *Commonsense 2001*.

Weld, D. S., Anderson, C. R., and Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proc. of AAAI 1998*, pages 897–904.

Yang, Q., Wu, K., and Jiang, Y. (2007). Learning action models from plan examples using weighted MAX-SAT. *Artif. Intell.*, **171**(2-3), 107–143.

Zhuo, H. H., Yang, Q., Hu, D. H., and Li, L. (2010). Learning complex action models with quantifiers and logical implications. *Artif. Intell.*, **174**(18), 1540–1569.