Learning action representations using kernel perceptrons

Kira M. T. Mourão

Doctor of Philosophy
Institute for Language, Cognition and Computation
School of Informatics
University of Edinburgh
2011

Abstract

Action representation is fundamental to many aspects of cognition, including language. Theories of situated cognition suggest that the form of such representation is distinctively determined by grounding in the real world. This thesis tackles the question of how to ground action representations, and proposes an approach for learning action models in noisy, partially observable domains, using deictic representations and kernel perceptrons.

Agents operating in real-world settings often require domain models to support planning and decision-making. To operate effectively in the world, an agent must be able to accurately predict when its actions will be successful, and what the effects of its actions will be. Only when a reliable action model is acquired can the agent usefully combine sequences of actions into plans, in order to achieve wider goals. However, learning the dynamics of a domain can be a challenging problem: agents' observations may be noisy, or incomplete; actions may be non-deterministic; the world itself may be noisy; or the world may contain many objects and relations which are irrelevant.

In this thesis, I first show that voted perceptrons, equipped with the DNF family of kernels, easily learn action models in STRIPS domains, even when subject to noise and partial observability. Key to the learning process is, firstly, the implicit exploration of the space of conjunctions of possible fluents (the space of potential action preconditions) enabled by the DNF kernels; secondly, the identification of objects playing similar roles in different states, enabled by a simple deictic representation; and lastly, the use of an attribute-value representation for world states.

Next, I extend the model to more complex domains by generalising both the kernel and the deictic representation to a relational setting, where world states are represented as graphs. Finally, I propose a method to extract STRIPS-like rules from the learnt models. I give preliminary results for STRIPS domains and discuss how the method can be extended to more complex domains. As such, the model is both appropriate for learning data generated by robot explorations as well as suitable for use by automated planning systems. This combination is essential for the development of autonomous agents which can learn action models from their environment and use them to generate successful plans.

Acknowledgements

Thanks are due to many people. First and foremost my supervisors, Mark Steedman, Ron Petrick and David Willshaw. Mark has been a source of insight, inspiration and motivation throughout and it has been a privilege to be his student. Ron has been a dedicated supervisor, always on hand with good advice and attention to detail, down to the last comma, be it midnight or morning, on holiday or at work. David has always encouraged me with his faith in my abilities, and I can only hope I have lived up to those expectations. I would also like to thank my examiners, Benjamin Kuipers and Subramanian Ramamoorthy, for the numerous corrections they proposed to the thesis: these have very much improved the final submission.

In addition, I am particularly grateful to Luke Zettlemoyer for many fruitful discussions, patient advice, enthusiasm, and overall his willingness to delve into the nitty-gritty of problems, both during his time as a post-doc in Edinburgh and later from Washington. I also benefited from meeting with many visiting researchers and would especially like to thank Ross Gayler for an enjoyable and enlightening discussion. The robot experiment was only possible thanks to Norbert Krüger's kind permission, and Dirk Kraft's time and patience in setting up the robot for the experiment, and running much of the data collection.

I have been lucky to have the support and companionship of other students in Mark Steedman's group, office mates past and present, and others, especially Catherine Ojo, Tom Kwiatkowski, Benjamin Rosman, Chris Geib, Emily Thomforde, Michael Auli, Lexi Birch, Prachya Boonkwan, Mark Granroth-Wilding, Christos Christodoulopoulos and Aciel Eshky. More broadly, the Neuroinformatics Doctoral Training Centre supported my work, more than just financially, but also through training, journal clubs, the write-up club and other ventures. In particular, no DTC thesis would ever have been completed without the efforts of Pat Ferguson, and latterly, Theresa Ironside, and I am grateful for their guidance through many administrative minefields. Thanks also to the EU Paco-Plus project which partially funded my research, both for the financial support and the opportunity to participate in the project.

Finally, Elson and my parents have always been in the background, there when I needed them. Alice contributed with her patient acceptance, and as a constant reminder of what AI still has to achieve.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Kira M. T. Mourão*)

To Mrs F

Table of Contents

Chapter 1

Introduction

A key aspect of theories of situated cognition is that an agent's internal representations are grounded in its sensorimotor experience of the real world. Action representations which are both grounded and yet capable of supporting high-level reasoning are therefore fundamental to cognitive processing such as planning, both in humans and cognitive agents. Such a grounded representation must derive from unreliable low-level interactions with the environment, yet provide sufficient structure to support reliable reasoning about when an action may be performed and what its outcome might be.

This thesis tackles the question of how to ground action representations, proposing an approach to learning action models in noisy, partially observable domains, using deictic representations and kernel perceptrons. Given a sequence of actions and structured observations of the world, the learning problem is to derive a set of STRIPS-like rules (Fikes and Nilsson, 1971) describing the preconditions and effects of each action. The approach presented here relies on deictic representations to structure the rule space, according to which a limited number of objects in the world are attended to. I consider both a simple algorithm which learns to predict action successes and corresponding effects in classical STRIPS domains, and a generalisation of this algorithm to learn in extended STRIPS domains. Additionally, I present a technique to extract STRIPS-like rules from the action models implicit in the original algorithms.

As an example of the learning problem in an extended STRIPS domain, namely the ADL Briefcase domain (Pednault, 1989), a sequence of actions and corresponding world states might be:

```
STATE:    (is-at home) (in book) (in phone) (at pen home) (at book home) (at phone home)
ACTION 1: (move home office)
STATE:    (is-at office) (in book) (in phone) (at pen home) (at book office) (at phone office)
ACTION 2: (take-out pen)
STATE:    (is-at office) (in book) (in phone) (at pen home) (at book office) (at phone office)
ACTION 3: (move office home)
STATE:    (is-at home) (in book) (in phone) (at pen home) (at book home) (at phone home)
```

The states describe the current world state in terms of the location of a briefcase and other items (either at home or the office), where the other items may be inside or outside the briefcase. Actions may move the briefcase, or insert or remove items. Some actions may not be possible; for example, Action 2 attempts to take the pen out of the briefcase, however, the preconditions of the action are unsatisfied since the pen is not in the briefcase. In this case, the action is assumed to be a no-op and the world state is unchanged. When an action is possible, the state of the world changes in accordance with the effects of the action.

The task of learning action preconditions and effects in this scenario is already beyond some approaches to learning action models, because of the indirect effect of the move action, which not only moves the briefcase, but also moves the items inside the briefcase. Additionally, an agent operating with unreliable sensors may take noisy or incomplete observations, resulting in it observing the sequence above as the following sequence of actions and observations:[1]

```
OBSERVATION: (in book) (at pen home) (at phone home) (is-at office) (not (in pen))
ACTION 1:    (move home office)
OBSERVATION: (is-at office) (in phone) (at book office) (not (is-at home)) (not (in pen))
ACTION 2:    (take-out pen)
OBSERVATION: (is-at office) (in pen) (at pen home) (at phone office) (not (at book office))
ACTION 3:    (move office home)
OBSERVATION: (is-at office) (at pen home) (in phone) (at book home) (not (at pen office))
```

The learning problem takes as input a sequence of actions and observations such as these and produces a description of the Briefcase action model. For example, the description specifies that the `(move ?m ?l)` action is successful if the briefcase is at

[1]Here observations include false properties of objects, to differentiate between properties of the world which are observed and those which are untrue. Whether to represent unobserved properties as untrue is a representational choice considered in Chapter 4.

location m, and has the effect of moving the briefcase and its contents from location m to location l. In PDDL (McDermott et al., 1998), this action has the form:

```
(:action move
  :parameters (?m ?l - location)
  :precondition  (is-at ?m)
  :effect (and (is-at ?l) (not (is-at ?m))
      (forall (?x - portable) (when (in ?x)
        (and (at ?x ?l) (not (at ?x ?m)))))))
```

1.1 Overview and motivation

This thesis proposes a new approach to learning relational action rules which can learn from noisy, incomplete observations. The algorithm learns both an implicit action model of the domain it is observing, and generates STRIPS-like rules from the implicit model, thereby providing domain descriptions suitable for use by automated planning systems.

1.1.1 Problem structure

The basis of the approach is the division of the learning problem into two parts: initially a classification method is used to learn an implicit action model, then explicit rules are derived from the resulting action representations. A secondary decomposition also underlies the learning process: a predictive model for each element of the state description is learnt separately, with the models only combined at the point of constructing full rule descriptions in PDDL format.

There are a number of reasons to decompose the problem in this way. By breaking the problem down into many simpler problems, it may be easier to solve. The effect of noise and partial observability is reduced, since by deriving explicit rules from a previously learnt implicit action model, the rule derivation can avoid the problems of learning from noisy or incomplete observations: the rules are now derived from a model which produces complete, noiseless training examples. Additionally, classification methods more commonly perform incremental learning from noisy and incomplete examples, while rule induction more often takes a batch approach. A structure which plays to these characteristics should have a wider choice of applicable pre-existing machine learning techniques.

1.1.2 Problem settings

The initial setting for the algorithm is action model learning in classical STRIPS domains, with the aim of generalising to more complex domains where some of the STRIPS assumptions are relaxed, in particular, the STRIPS scope assumption that all objects which appear in the preconditions or effects of an action also appear in the argument list of the action. The formulation of the STRIPS learning problem as a classification problem is straightforward, as states can be represented by vectors in a simple attribute-value representation. The vector representation facilitates a fundamental process of classification: comparisons of different states in order to identify regularities in the sequence of actions and observations.

A challenge for generalising classification to settings without the STRIPS scope assumption is that, when working with general relational representations, these comparisons rapidly lead to computational inefficiency due to the many possible ways to match objects across different states when performing a comparison (Haussler, 1989). The key step in this thesis is to formulate the state representations for both settings in terms of deictic references. A deictic reference is a pointer to objects which have a particular role in the world, with object roles coded relative to the agent or current action (Agre and Chapman, 1987). This leads to a graphical representation of states in terms of deictic references and a novel graph kernel which uses deictic references to limit the number of object matches in comparisons.

1.1.3 Rule extraction

The two-stage learning approach means that post-processing is required to extract STRIPS-like rules from the implicit learnt models. At this juncture it would be possible to apply an existing method for learning action dynamics to learn an explicit action model from the implicit model, as observations can now be complete and noiseless. However, this fails to take advantage of the structure already learnt and so is likely to be inefficient. Similarly it would be possible to apply standard rule extraction techniques to each classifier predicting some element of the state, followed by a process to collate the rules into full action descriptions. Since these methods tend to produce large numbers of rules and are often computationally intractable, this was also rejected.

Instead, a new method for rule extraction was developed with the goal of re-using some of the structure already uncovered by classification learning. Rule extraction is essentially a guided search through the space of possible rules, while the structure discovered by a voted perceptron classifier is encoded in its support vectors. The rule extraction method combines these by using the support vectors to seed the search for rules in an approach based on both rule extraction and feature selection methods. A final rule combination step heuristically combines the individual rules into full action descriptions. The resulting rules compare favourably to implicit models, with no statistically significant differences between predictions using the rules or the models. The method as presented only applies to classical STRIPS domain models, but there is scope to extend it to include conditional effects, disjunctive preconditions, universally quantified effects and possibly also to include probabilistic effects.

1.2 Contributions

In summary, the key contributions of this thesis are:

- a new approach to learning relational action models, using a two-stage learning process which first learns an implicit action representation and then processes the representation to form STRIPS-like action rules;
- a novel graph kernel based on deictic references;
- a novel rule extraction method; and
- a formulation of state representations in terms of deictic references.

1.3 Thesis outline

The remainder of this thesis is structured as follows. Chapter 2 covers the background to the problem and reviews previous work on learning action models. Chapter 3 sets out definitions and notation used in the thesis, and provides details of the training and testing datasets. Chapter 4 covers learning of implicit STRIPS models in partially observable and noisy domains, and Chapter 5 extends the approach to more complex models. In Chapter 6 a method is given to extract explicit formal STRIPS rules from implicit models of the type learnt in Chapter 4, with the potential to extend the method to the models of Chapter 5. The final chapter concludes with a summary of the thesis and a discussion of possible future work.

Chapter 2

Affordance learning in context

In this chapter I introduce affordances and how they relate to action model learning. I discuss some of the different action formalisms available and cover previous work on both grounding action models and relational learning of action models. Finally, I consider the different representations used and the effects of the choice of representation on computational tractability.

2.1 Affordances

Learning domain dynamics by exploration naturally leads to learning the action possibilities, or affordances, available to an agent in a domain. Gibson (1979) defined an affordance as

> a resource or support that the environment offers an animal for action. The animal must possess the capabilities to perceive and act on it.

This can equally apply to any agent in an environment. The affordances of a situation vary between agents and depend on the agent's physical attributes, and sensory and motor abilities. An apple affords eating or perhaps throwing or kicking, for a human, but not for a cat. In different situations the affordances related to an object may vary: an apple will not afford eating if full of maggots.

Gibson proposed that affordances are the basis of perception in animals, intimately linking perception, action and the external environment. Defining an agent's interaction with the world in terms of affordances can lead to a common representation for actions and objects which can be shared among agents with the same abilities. Additionally, if an agent perceives a set of affordances for a situation, it can also advance a prediction about the outcome of performing an afforded action.

An affordance predicts an action on an object which will have some effect on the environment. Usually the effects of an action are included in the affordance definition. For example, Şahin et al. (2007) incorporate the effects of the behaviour, defining an affordance as a relation between an entity in the environment and a behaviour of the agent, such that applying the behaviour to the entity results in a particular effect. Similarly, Steedman (2002) includes effects by formulating actions as functions from preconditions to postconditions. The affordances of an object are then defined to be the set of such actions which can be carried out on the object by an agent.

There is a good reason to include effects of actions within affordances: when we learn the affordance relation by performing an action in the world, we also have the effect information available. Since the afforded action must have some effect on the environment (or at least the agent must believe[1] there is some effect), learning an affordance involves at least detecting that there was an effect. It seems implausible that an effect of an action would be detected but not associated in some way with the action, so in this thesis, affordances will be defined to include both the actions afforded by a situation, and the effects of performing those actions.

2.2 Action formalisms

To represent affordances, we can look to the many action formalisms developed for AI in the planning, common-sense reasoning and intelligent agents subfields. Van Otterlo (2009) divides action formalisms into *action languages* and *action logics*. Action languages focus on compactly representing and modelling actions to support computationally tractable planning. In contrast, the more formal action logics incorporate the actions as axioms within the logic, supporting reasoning and inference on the actions.

Common to most single agent action formalisms is the expectation that actions take place when the world is in some fixed state or situation, that actions are atomic (instantaneous), blocking (only one can occur at a time), and are the only cause of change in the world. Change is seen in the changing values of predicates describing the world: *fluents* are predicates which may change as a result of some action, while *static predicates* are predicates which never change. There is also the notion of an action

[1]In this thesis I do not consider belief-based learning, where an agent learns while taking into account its own and other agent's beliefs, and competes with other agents for resources within the world. Instead here the agent does not consider and is unaware of the actions of other agents, and does not differentiate between its own beliefs about the state of the world and the actual state of the world, similar to standard reinforcement learning.

precondition, which must be satisfied in order for an action to take place, and an action effect, specifying the results of an action in the world (Schwind, 1999; Van Otterlo, 2009). Since the preconditions determine if an object affords particular actions in a particular state, most action formalisms could be used to represent affordances.

2.2.1 Fundamental issues for any formalism

There are fundamental problems any action logic or language must handle. These relate to how the action formalism supports tractable reasoning about the world (the frame problem and associated qualification and ramification problems), and how the formalism links its representation to the world (the symbol grounding problem, the binding problem and the correspondence problem). These are discussed in detail below.

2.2.1.1 The frame problem

The frame problem is the problem of describing both the effects and non-effects of actions without requiring an exhaustive description of every possible effect or non-effect of every action (McCarthy and Hayes, 1969). Some means of determining non-effects of actions is needed, otherwise it would be necessary to re-evaluate the state of the world after every action, since it would not be known what else might have changed besides the immediate effects of the action. The frame problem creates both representational and computational issues. Representationally, in a complete description of the world, it can rapidly become impractical to represent vast numbers of frame axioms describing non-effects of actions. Computationally, it is infeasibly slow to make inferences from situations described mostly by large numbers of non-effect axioms. In the context of a robot learning outcomes using a full description of the state of the world, the frame problem is present in the sense that we have a set of exhaustive descriptions of the world both before and after any action, and must generate the appropriate set of rules relating prior and following states: the rules must deal with the frame problem in both representational and computational aspects.

2.2.1.2 The ramification problem

Related to the frame problem, the ramification problem (Finger, 1987) concerns how to represent and reason about the *indirect* effects of an action, arising as a consequence of the direct effects of the action, and of the rules governing the world itself. For example, in the Briefcase domain, when a briefcase is moved from one location to another, all the objects inside the briefcase also move to the new location. The direct effect of the briefcase changing location will be encoded in the action description, but the indirect effect of the briefcase contents changing location may be derived from known causal relations between the fluents. Alternatively, the indirect effects could be encoded as direct effects, but this can lead to very large action descriptions.

2.2.1.3 The qualification problem

The qualification problem (McCarthy, 1977) is also related to the frame problem. Outside artificial domains, there are vast numbers of actual preconditions (qualifications) of an action (although most of the time, most of them will be true). For instance, McCarthy discusses the "potato-in-the-tailpipe" scenario, where the action of starting a car has the precondition that the key is in the ignition, but the success of the action is also qualified by the preconditions that the car has petrol, the battery is connected, there is no potato in the exhaust pipe, and any number of other possibilities. Checking every possible qualification on the action is infeasible, and for most preconditions, unnecessary.

2.2.1.4 The symbol grounding problem

The symbol grounding problem (Harnad, 1990) is related to the problem of assigning meaning to arbitrary symbols. Some symbols can be defined in terms of other symbols, but at least some symbols must be defined in some other way, or *grounded*, in order to avoid infinite regression. In this project, the intention is to ground representations of affordances in terms of associations between perceptual features and actions. If objects are represented by their affordances then this grounds object representation in sensorimotor experience — except that the representations of perceptual features and actions themselves need to be grounded.

2.2.1.5 The binding problem

The binding problem (Treisman, 1998) is the problem of representing features in such a way that (exactly) those features corresponding to the same entity are bound together. In terms of learning affordances of objects, binding will be needed to understand which features correspond to which objects. The present thesis assumes that perceptual binding is a property of the perceptual system, occurring before learning takes place.

2.2.1.6 The correspondence problem

The correspondence problem (Aggarwal et al., 1981) is the problem of determining, over the course of several observations, which object in one observation corresponds to which object in another observation. Although usually considered a vision problem, a solution is necessary for any action formalism which depends on discrete observations of the world before and after an action. Correspondence may be achieved by matching objects which have the same features but this can run into problems when there are many similar or identical objects in a scene, or when observations are unreliable. This thesis assumes that the perceptual system resolves the correspondence problem before learning takes place.

2.2.2 STRIPS and related action languages

In this thesis, actions will be learnt as operators in the STRIPS (STanford Research Institute Problem Solver) action language (Fikes and Nilsson, 1971), and its extensions. In classical STRIPS, domains are assumed to be deterministic and states are fully observable. The state description is symbolic, with states represented by conjunctions of ground predicates. Predicates not explicitly mentioned in the description are assumed to be false (*closed world assumption*) and so states can be fully described by conjunctions of only positive literals.

STRIPS operators consist of an action name, a set of preconditions, an add list and a delete list. Preconditions are a conjunction of positive literals which must hold for the action to be performed. The add list is a set of fluents whose values are true after the action, and the delete list is a set of fluents whose values are false after the action. Together the add and delete lists encode the effects of the action. Any fluent not mentioned in the description of the action stays unchanged (the *STRIPS assumption* (Waldinger, 1977)). Objects mentioned in the preconditions or the effects must be listed in the action parameters (the *STRIPS scope assumption* (Walsh and Littman, 2008)). Finally, actions always succeed if their preconditions are satisfied (Russell and Norvig, 2009).

STRIPS was subsequently extended to ADL (Action Description Language) (Pednault, 1989). ADL is more expressive than STRIPS, while also relaxing some of the associated constraints. The language is extended to include an equality predicate and types. Actions may have negative preconditions, which in turn allows both open or closed world reasoning. Preconditions may also have disjunctions, and both preconditions and effects may include quantifiers. Actions may also have conditional effects, where additional effects may occur as the result of an action, if secondary preconditions are satisfied in addition to the main preconditions of the action.

PDDL (Problem Domain Description Language) (McDermott et al., 1998) is a standardisation of action languages, including STRIPS and ADL as subsets. Originally devised to provide a common planning language for the International Planning Competitions (IPCs)[2] it is much more expressive than STRIPS or ADL, supporting, for example, derived predicates, and notions of time and duration.

The STRIPS assumption provides a solution to the frame problem,[3] as it avoids explicitly enumerating fluents which are unchanged by an action. ADL makes a similar assumption (Pednault, 1989). The qualification and ramification problems are only resolved by requiring that any necessary precondition is explicitly listed, and that indirect effects must be explicitly coded as direct effects.

2.3 Representation

An action precondition is a concept to be learnt. There are a variety of increasingly more complex representations which could be chosen for a precondition. De Raedt (2008) defines a hierarchy of representations which may be used in relational learning, ranging from propositional to logical.

At its simplest, a precondition may be represented by a set of propositions, Boolean attributes which take the value *true* or *false*. In this *Boolean representation*, only *true* attributes need to be specified, since the remainder will then be *false*. For instance, the example in Figure 2.1 could be represented as:

{`clear-B, B-on-C, C-on-table, clear-D, D-on-table`}.

A more expressive propositional representation is an *attribute-value (AV) representation* where attributes need not be Boolean-valued but could also take discrete or continuous values. For instance, the example in Figure 2.1 could be represented as:

{`colour-B=blue, colour-C=blue, colour-D=red,`
`clear-B, B-on-C, C-on-table, clear-D, D-on-table`}.

Attribute-value (AV) representations can be reduced to Boolean representations if the domains of the attributes are discrete. However, AV forces each attribute to take a single value whereas the corresponding Boolean representation permits an attribute to

[2]See `ipc.icaps-conference.org`.
[3]When states are described using only atomic formulae.

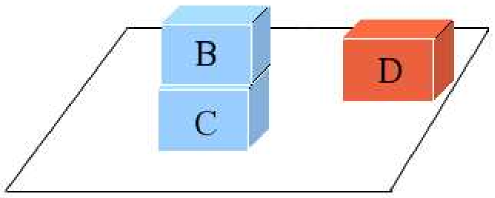


Figure 2.1: BlocksWorld state example

take multiple values. For example, {`colour-B=blue`} in an AV representation translates to a proposition {`colour-B-blue`} in a Boolean representation, but the Boolean representation would also permit {`colour-B-blue, colour-B-red`}.

The propositional representations are not particularly suitable as representations of action preconditions, since there is little support for relations. A more natural representation would be a *relational representation* which, in contrast to the propositional approaches, allows both objects and relations between objects to be represented. For instance, the example in Figure 2.1 could be represented as:

{`clear(B), on(B,C), ontable(C), ontable(D), clear(D),`
`colour(B)=blue, colour(C)=blue, colour(D)=blue`}.

A relational representation can be reduced to an AV representation but it requires taking the cross-product of the relations to produce a single *universal relation* enumerating all possible relations in the world. Also, relational representations can be extended to full-blown logical representations with functors and structured terms: this type of representation is beyond the scope of this thesis.

In concept learning, most machine-learning methods have focused on attribute-value learning. Grounded approaches to action model learning also tend to be AV, because the problem of learning relations is still to be addressed, and because the observed world features are naturally limited by the restricted amount of data available from sensors. In contrast, relational approaches usually use relational representations.

2.3.1 Biases and related complexity results

Concept learning approaches may employ bias to constrain the search space of possible hypotheses. A common source of bias is to restrict the possible form a hypothesis may take, reducing the size of the hypothesis space, for example, by permitting only conjunctive concepts, k-DNF or k-CNF. Object types may be introduced, which then restrict the potential arguments of actions and fluents by only allowing particular types, again reducing the size of the hypothesis space. Additionally the system may be endowed with a preference bias towards certain types of hypotheses, often shorter (and therefore simpler) hypotheses.

The type of bias affects the tractability of learning. For instance, it is an open problem in learning theory whether DNF expressions can be learnt efficiently. Since kernel perceptrons will form the basis of the learning approach in this thesis, it should be noted that it is known that the kernel perceptron cannot efficiently either learn or PAC-learn DNF (Khardon et al., 2005).

All of these types of bias are used in the action learning literature. The robot-based attribute-value learners typically assume conjunctive concepts, and this is generally enough to make learning tractable since the set of attributes is relatively small. Relational STRIPS learners will, by definition, assume conjunctive preconditions (and effects) (Wang, 1995; Yang et al., 2007; Walsh and Littman, 2008).[4] More complex domains require more expressive hypotheses, so other learners assume hypotheses which are small disjunctions of conjunctions (Rodrigues et al., 2010b), possibly with an additional preference for shorter hypotheses (Benson, 1996; Pasula et al., 2007). Some support quantifiers (Rodrigues et al., 2010b; Zhuo et al., 2010), logical implication (Zhuo et al., 2010) or CNF (Amir and Chang, 2008).

In concept learning, with fully-observable, noiseless examples, conjunctions and k-DNF in attribute-value representations are PAC-learnable (Valiant, 1984, 1985). However, finding a consistent hypothesis of minimum size is NP-hard (Haussler, 1988). More recently, it has been shown that conjunctions and k-DNF are PAC-learnable under partial observability where attributes are masked independently (Decatur and Gennaro, 1995) or arbitrarily (Michael, 2007). Furthermore, conjunctions and k-DNF are PAC-learnable with combined classification noise and attribute noise, or both classification noise and partial observability (for k-DNF the noise levels must be known) (Decatur and Gennaro, 1995).

Similarly, in deterministic, fully observable, noiseless domains, Walsh and Littman (2008) show that STRIPS action models can be learnt efficiently, provided that preconditions have at most k fluents. This restriction is due to a problem of exploration rather than of learning: if a precondition is a conjunction of instances of all P predicates in a world, then simply finding the state where the precondition holds, in order to generate a positive example to learn from, can take in the worst case $O(2^P)$ steps. Clearly, learn-

[4] Wang (1995) also learns conditional effects and negative preconditions.

ing action models in other languages must be subject to the same constraint, so even if there were an efficient DNF learner available, only preconditions in k-DNF could be learnt efficiently.

Conversely, with a relational representation, conjunctions (and therefore k-DNF) are neither learnable nor PAC-learnable, at least where the learner produces hypotheses which are also existential conjunctive concepts (Haussler, 1989). This holds even when the instance space is severely restricted with only unary relations, Boolean-valued attributes and where every example consists of only two objects. Haussler (1989) notes that the difficulty is with the ambiguity introduced by the lack of a fixed mapping between objects in different examples: the different hypotheses considered by the learner must account for each of the possible mappings.

These results suggest that it may be possible to achieve tractable learning of pure and extended STRIPS models even in partially observable, noisy domains. However, it is unreasonable to expect tractable learning of more general models without significant further constraints.

2.3.2 Rule search

Concept learning is often characterised as a search through the space of possible hypotheses for a concept which applies to a set of examples (Mitchell, 1982). The search can be aided by structuring the space as a lattice ordered by a generalisation relation on the hypotheses. However, the nature of the search depends on how the hypotheses and examples are specified, and the choice of generalisation relations.

The learning problem may be modelled (De Raedt, 2008) in terms of:

- a language of examples $\mathcal{L}_e$ consisting of descriptions of examples or observations,
- a language of hypotheses $\mathcal{L}_h$ consisting of hypotheses about examples or observations, and
- a covering relation $c : \mathcal{L}_e \times \mathcal{L}_h$ which determines if a hypothesis matches an example.

The choice of languages and covering relation gives rise to different learning settings. Two common settings are *learning from entailment* and *learning from interpretations* (De Raedt, 2008). In learning from interpretations, $\mathcal{L}_e$ is a set of interpretations, usually Herbrand interpretations (informally, true ground facts which fully describe a possible observation or situation), while $\mathcal{L}_h$ is a set of logical formulae. For an example $e \in \mathcal{L}_e$ and hypothesis $H \in \mathcal{L}_h$, H covers e iff e is a model of H. For instance, in a BlocksWorld domain if the example $e \in \mathcal{L}_e$ is the interpretation

$$\{UNSTACK(A,B) \wedge ON(A,B) \wedge ONTABLE(B) \wedge CLEAR(A) \wedge ARMEMPTY\}$$

and $H \in \mathcal{L}_h$ is

$$UNSTACK(x,y) \text{ :- } ARMEMPTY \wedge ON(x,y)$$

then H covers e since for any θ chosen from the ground substitutions $\{x/A, y/B\}$, $\{x/B, y/A\}$,$\{x/A, y/A\}$ and $\{x/B, y/B\}$, $H\theta$ is always true in e.

In learning from entailment, $\mathcal{L}_e$ is a set of clauses (usually definite) while $\mathcal{L}_h$ is a set of theories (sets of clauses). For an example $e \in \mathcal{L}_e$ and hypothesis $H \in \mathcal{L}_h$, H covers e iff $H \models e$ (H and $\neg e$ is unsatisfiable). For instance, in BlocksWorld again, if e is

$$\{UNSTACK(A,B), ON(A,B), \neg ONTABLE(B)\}$$

and H is

$$UNSTACK(x,y) \text{ :- } ON(x,y)$$

then H covers e.

Learning from interpretations implicitly assumes that each example is completely specified, a form of closed world assumption, and so it does not support incomplete examples. The setting of learning from partial interpretations has been considered, where a partial interpretation is a set of true or false ground facts describing a situation, and where some facts may be unknown. This setting does support incomplete examples, but it is equivalent to learning from entailment (De Raedt, 1997). Since learning from interpretations supplies a full set of facts about an example, much more information than learning from entailment, learning from interpretations is an easier and more tractable setting to work in.

Given a lattice of hypotheses in either the learning from interpretations or learning from entailment settings, the search for a concept is structured by deciding the covering relation between examples and hypotheses, and eliminating unsuitable hypotheses. Usually an approximation to logical entailment is used in the form of θ-subsumption (Plotkin, 1970), which approximates $c_1 \models c_2$ by a test for the existence of a substitution θ such that $c_1\theta \subseteq c_2$. However, θ-subsumption is NP-complete (Kapur and Narendran, 1986). OI-subsumption (subsumption under Object Identity) restricts θ-subsumption by forcing each term of a clause to be different. It therefore simplifies the structure of the search space by preventing substitutions where several terms can be collapsed into one. It is also strictly weaker than θ-subsumption. Computationally, OI-subsumption is equivalent to graph isomorphism, which is neither known to be NP-complete nor has a known polynomial algorithm. In practice, however, OI-subsumption is more efficient that θ-subsumption. Recently Rodrigues et al. (2010a) have learnt action rules using the learning from interpretations setting and OI-subsumption.

2.4 Modelling abstractions

Learning action models from experience amounts to estimating a (possibly stochastic) transition function[5] from observations. Such transition functions can be learnt in the context of state space models such as Markov Decision Processes (MDPs) (Puterman, 1994), and generalisations thereof. As discussed below, learning transition functions in the context of MDPs corresponds to learning in propositional, fully observable domains. While techniques exist to learn transition functions in this case, once we move to relational and/or partially observable domains, the available techniques are much more limited.

2.4.1 Markov Decision Processes

MDPs are a combination of states, actions, a transition function between states, and a reward function. Actions control when the system changes from one state to another, with the resulting state determined by the transition function. The reward function assigns rewards to particular states, or performing particular actions in some states, and is used to drive learning towards particular goals. A crucial assumption is that the system is *Markovian*, that is, the result of an action depends only on the current state, and not on previous actions or states (the Markov property). Thus knowledge of the sequence of actions and states leading up to the current state is not necessary to make an optimal decision about which action to take. Formally, MDPs are defined as below.

[5]In this thesis I assume the transition function is deterministic (the possibility of extending to stochastic transition functions is discussed in Section 6.5.3).

Definition 2.4.1. A *Markov Decision Process (MDP)* is a tuple (S, A, T, R) where:

- S is a finite set of states;
- A is a finite set of actions;
- $T : S \times A \times S \rightarrow [0,1]$ is a transition probability function, where $T(s,a,s')$ is the probability of a transition occurring between state s and state s' via action a; and
- $R : S \times A \rightarrow \mathbb{R}$ is the expected reward function, where $R(s,a)$ is the expected reward for performing action a in state s.

Now learning the preconditions and effects of actions in a fully-observable world amounts to learning the transition function T of an MDP. There are established algorithms for learning the transition function in MDPs, for example, the EM-based algorithm of Murphy (2002). However, a major issue with using MDPs in the relational setting is that MDPs require an explicit enumeration of the state space. Even in a small BlocksWorld with 5 blocks, the number of grounded fluents is 41, requiring a transition matrix with (an infeasible) $(2^{41})^2$ entries. Additionally, in an MDP it is assumed that the states of the world are fully observable: an assumption which does not hold for the scenarios considered in this thesis. The MDP abstraction has been extended in a number of ways to deal with partial observability, large state spaces and relational data. Below I discuss these extensions in turn.

2.4.2 Partially observable MDPs

If state observations are incomplete, the world can be modelled by a partially-observable MDP (POMDP) (Kaelbling et al., 1998). In a POMDP the agent cannot observe the actual state, but only has an observation of the state, which may be obscured, or corrupted by noise. This is modelled using a space of possible observations in combination with an observation probability function, which determines the probability of an observation given a state and an action. Formally:

Definition 2.4.2. A *Partially Observable Markov Decision Process (POMDP)* is a tuple (S, O, A, B, T, R) where S, A, T and R are as for MDPs above, and:

- O is a finite set of observations; and
- $B : O \times A \times S \rightarrow [0,1]$ is an observation probability function, where $B(o,s,a)$ is the probability of observation o occurring when action a is performed and the resulting state is s.

In place of knowledge about the state, the agent now only has available a *belief state*, namely a probability distribution over the set of states, indicating the agent's belief about its current state. The belief state can be updated after an action and an observation by conditioning on the observation, and using the transition and observation probability functions. The increased complexity of POMDPs also makes learning of the transition model much more complex. For instance, Shani et al. (2005) simultaneously learn an underlying model and a POMDP solution, however the approach is limited to small domains.

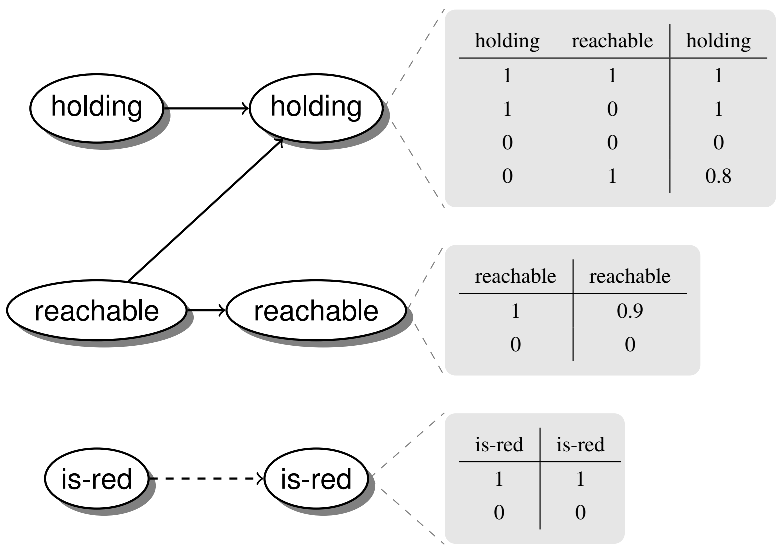
2.4.3 Factored MDPs

In the description so far, states have been modelled as monolithic entities, but often states will have structure which can be exploited. Furthermore, intuitively the success of an action typically depends on a few aspects of the state and not on others. For instance, the success of a grasping action will usually depend on the current state of the gripper and the object to be grasped, but not on the locations or colours of other objects in the world. Factored MDPs (Boutilier et al., 1995) or factored POMDPs (Boutilier and Poole, 1996) address this by modelling the state as a set of separate factors. Formally:

Definition 2.4.3. A *factored MDP* is a finite MDP where the set of states S is described by a set of n random variables $X = \{X_1, \ldots, X_n\}$, so that each $s \in S$ is a cross-product of *factors* $x_1 \times x_2 \times \ldots \times x_n$ where $x_i \in X_i$. Each X_i has a finite set of possible values. *Factored POMDPs* are defined analogously.

As before, the associated transition matrix is infeasibly large, with size exponential in n. However, factorisation means that the transition function may be compactly modelled, for example, using Dynamic Bayes Nets (DBNs) (Dean and Kanazawa, 1989), subject to the assumption that each factor depends only on a small number of factor values in the previous timestep.

The DBNs are defined as follows. The transition model is described by a separate DBN for each action a. Each DBN is a two-layer directed acyclic graph with nodes $\{\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}\}$ and $\{\mathrm{X}'_1, \ldots, \mathrm{X}'_\mathrm{n}\}$, where X_i denotes the random variable X_i at time t, and X'_i denotes the random variable X_i at time $t+1$. Edges may only link nodes in $\{\mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}\}$ to nodes in $\{\mathrm{X}'_1, \ldots, \mathrm{X}'_\mathrm{n}\}$. The set of parents of the node X'_i in the graph corresponding to action a is denoted by $Parents(\mathrm{X}'_\mathrm{i}, a)$. Each node X'_i has an associ-



holding	reachable	holding
1	1	1
1	0	1
0	0	0
0	1	0.8

reachable	reachable
1	0.9
0	0

is-red	is-red
1	1
0	0

Figure 2.2: An example DBN for a grasp action in a simple robot domain (see text). Conditional probability tables associated with each node at time $t+1$ are shown in grey. The value of the `holding` predicate depends on the values of the `holding` and `reachable` predicates in the previous timestep. The `grasp` action is not always successful, even if the object is `reachable`. The value of the `reachable` predicate only depends on the previous value of the `reachable` predicate, but may change in response to a `grasp` action, which could push the object out of reach instead of grasping. The value of `is-red` does not change and so only depends on the previous value of `is-red`.

ated conditional probability table (CPT) $Prob(\mathrm{X}'_\mathrm{i}|Parents(\mathrm{X}'_\mathrm{i}, a))$. The state transition probabilities are given by $P(X'|X,a) = \prod_{i=1}^{n} Prob(\mathrm{X}'_\mathrm{i}|Parents(\mathrm{X}'_\mathrm{i}, a))$.

For example, consider a toy domain where a robot can perform a grasp action in a world with one object. The domain has predicates to indicate whether the robot is holding the object (denoted by `holding`), whether the object is reachable by the robot (denoted by `reachable`) and the colour of the object (denoted by `is-red`). The result of the grasp action depends on whether the robot is already holding the object and whether the object is reachable, but not on the colour of the object. Even if the object is reachable, the grasp may sometimes fail. This situation is described by the DBN shown in Figure 2.2. In this example, it is likely that `reachable` is in fact dependent on `holding` but this would break the assumption that edges may only link nodes between timesteps, and so is omitted.

The problem of learning the transition function by finding a DBN which best fits some data set is the problem of *structure learning*. The structure learning problem for

DBNs is NP-hard, however, recently approximate approaches have been developed for structure learning in factored MDPs (Jonsson and Barto, 2007; Strehl et al., 2007; Diuk et al., 2009; Chakraborty and Stone, 2011). One difficulty with using such approaches to learn transition functions in relational domains is that the DBN representation requires a node for each changing attribute *of each object*, hampering generalisation. Additionally the relational predicates are often inter-dependent, breaking the assumption that edges only link nodes between timesteps. As a consequence, these approaches are not directly applicable to relational domains. Relational generalisations of the DBN representation exist but have had limited application to the problem of learning transition functions (see PRMs in Section 2.4.4 below).

2.4.4 Relational (PO)MDPs

Beyond factored MDPs, first-order logical representations for MDPs have been proposed, such as the Independent Choice Logic of Poole (1997); Probabilistic Relational Models (PRMs), a relational analogue of DBNs (Getoor et al., 2007); or representations based on the situation calculus (an action logic, Section 2.2) (Reiter, 2001; Boutilier et al., 2000). With regard to partial observability, the latter representations have recently been extended to partially observable domains, to give relational POMDPs (Wang and Khardon, 2010) or first-order POMDPs (FO-POMDPs) (Sanner and Kersting, 2010). However, methods which operate within these frameworks to learn the underlying transition model have yet to be developed. Restricting ourselves to the fully observable case, transition functions encoded by PRMs may be learnt via algorithms which extend structure learning for DBNs (Getoor et al., 2007). However, although PRMs have been used to represent states in relational MDPs, it was assumed that the relations between objects never change (Guestrin et al., 2003). For the types of planning domains we consider, this assumption is clearly unrealistic. In summary, although methods exist to learn transition functions in MDPs, and to some extent in POMDPs and factored MDPs, these methods do not translate to learning in relational, noisy, partially observable domains.

2.5 Previous approaches to action learning

The literature on learning action models divides into two main approaches. First, learning grounded action models by constructing transition rules from actions and robot sensor data coded as sets of objects or raw sensor readings, and predicates derived from this data. Second, learning relational action models by working within the space of transition rules and attempting to exploit relational structure in order to improve speed and generalisation performance. Both of these approaches are discussed in detail below.

2.5.1 Learning grounded action models

A number of lines of research have been directed towards grounding objects, attributes, relations and actions in sensorimotor experience. Typically this involves a robot equipped with a variety of sensors, each producing a stream of continuous values. The robot's observations will be partially observable and noisy, due to either or both its environment and limitations of its sensors. The task is to derive symbolic properties of the world, from objects and attributes to descriptions of actions and relations.

Some approaches do not extend as far as identifying objects[6] in the world, operating instead on holistic experiences defined by particular groupings of sensor values (Schmill et al., 2000; Doğar et al., 2007). The actions available to the robot are assumed to be known, and sensor readings before and after an action are recorded. The readings may be a snapshot (Doğar et al., 2007) or a series (Schmill et al., 2000). Broadly, sensor readings after an action are clustered into sets of similar "experiences" or effects which are then associated with sensor readings before the action (e.g., using decision trees (Schmill et al., 2000) or feature selection and SVMs (Doğar et al., 2007)). The result is a rudimentary operator model which can be used to form simple plans.

A related approach is taken by Holmes and Isbell (2005) who also learn and refine operators which predict (individual) sensor values based on current sensor readings and a known action. In addition, when a prediction fails for some operator, they create synthetic sensors corresponding to posited hidden or derived variables whose values, if known, correspond to whether the operator would predict correctly or not. Although the agent cannot observe the values of synthetic sensors, it can learn to predict their values, thereby improving the reliability of its operators.

[6]Without objects, attributes and relations are not identified either.

More sophisticated approaches are able to extract notions of objects and attributes from sensor data. One extensive line of research has demonstrated how an autonomous agent may learn the structure of its sensors (Pierce and Kuipers, 1997; Olsson et al., 2006), identify objects in its environment and discover attributes of those objects (Modayil and Kuipers, 2004), and learn how the world changes in response to the agent's actions (Modayil and Kuipers, 2007, 2008).[7] Here objects were identified by comparing sensor readings to a previously constructed static world model, and tracking clusters of readings (in time and space) which were unaccounted for by this model. Other approaches to identify objects are also based on the principle that movement can be used to segment objects, by coordinating a visual system with manipulation (Metta and Fitzpatrick, 2003; Kraft et al., 2010). The combination of the concept of an object and sensor data which can be attributed to it permits derivation of simple features directly from the sensor readings, for example, the distance to or position of the object at some point in time. More complex features such as shape may be constructed over time from multiple sensor readings (Modayil and Kuipers, 2008; Kraft et al., 2010).

Once objects and their features are available, the problem of learning how object features change in response to actions is an extension of the problem of learning global changes in the world as a result of actions. The same principles apply. Firstly, regions in variable space which may be considered to be the same or different are identified by using clustering approaches to partition the space. It is then possible to identify when an action changes a variable, in the sense of moving its value from one partition of the space to another (Modayil and Kuipers, 2007, 2008; Montesano et al., 2008). Very recent work additionally dispenses with the assumption that available actions are known, requiring instead only knowledge of motor primitives (Slowinski and Guerin, 2011; Mugan, 2010). By treating the execution of a motor primitive as a variable in its own right, the changes associated with each motor primitive can be learnt, followed by increasingly complex hierarchical definitions of actions grounded in the motor primitives. In a different vein, Mukerjee (2009) gives an unsupervised learning method for both discriminating actions and determining their parameters using video sequences. The method operates by clustering temporal data selected via an attentional mechanism in the form of a saliency map. It was able to extract actions and parameters from videos of interacting agents, and to differentiate between the different roles of the action parameters (e.g., the agent following and the agent being followed).

To date the most comprehensive approach to grounded learning from sensorimo-

[7]As before, the actions available to the robot are assumed to be known.

tor experience is the QLAP algorithm (Mugan, 2010). QLAP (Qualitative Learner of Actions and Perception) assumes as input a set of continuous variables describing the environment, generated from the environment by a perceptual function or data factoring process (e.g. Modayil and Kuipers, 2007). Outputs are primitive motor actions which are converted to raw motor variables by a motor conversion process (e.g. Pierce and Kuipers, 1997). QLAP discretises the continuous inputs by identifying *landmarks*, points which meaningfully divide the continuous space of values. The set of landmarks is recursively extended as actions are identified and refined. QLAP identifies *contingencies* in the discretised description of the environment, where a change in a variable (across a landmark point) is closely followed by change in another variable (typically the first change will relate to a motor action). This change is encoded as a DBN where the first change is encoded in the first time slice, and the second change in the second time slice. QLAP then seeks additional context variables which may improve the prediction of the DBN.

By creating DBN representations of actions, QLAP can convert the DBNs into MDPs and use standard MDP learning techniques to generate policies and plans. Rather than modelling the world as a single MDP, this models the world using many smaller MDPs, reminiscent of the options framework. These MDPs may additionally be linked together to form larger plans using goal-regression planning.

QLAP had been used exclusively on non-relational domains. Part of the reason for this is that QLAP does not support learning of relational predicates, but ultimately the underlying substrate of QLAP is propositional and not relational. For instance, even if QLAP were provided with a predefined relational domain, contingencies would be learnt from the full grounded state description. The problems with structure learning in MDPs on grounded relational domains also apply here: the DBNs require a node for each changing attribute on each object, making it difficult to generalise across states. Similarly, even if relational contingencies were learnt, the relational counterparts of DBNs and MDPs (e.g. PRMs and RMDPs) would be needed in order to model the domain dynamics.

In summary, existing work has shown how a robot may autonomously acquire knowledge of its sensors and motion primitives, as well as of the external environment in terms of objects and their attributes. Rules governing how the robot's actions cause changes in its environment can also be learnt, even when the environment is noisy or partially observable. However, none of the existing work on grounded models has considered how to learn relations between objects, let alone how relations change as a result of actions. In general, learning grounded relations is an important open problem, which recent studies are only just beginning to address (Rosman and Ramamoorthy, 2011; Sjöö and Jensfelt, 2011), and only in the context of supervised learning. Given this existing body of work, in this thesis it will be assumed that a world representation can be obtained consisting of symbols for objects, their attributes and relations, attributes of the world, and actions. It is however clear that data derived from a grounded representation of the world will be noisy and incomplete, even for data at the level of objects and relations rather than at the level of sensor readings.

2.5.2 Learning relational action models

Most previous work on learning relational action models makes similar assumptions to those discussed above. However, much work also relies on the provision of prior knowledge of the action model. For example, strategies include seeding initial models with approximate planning operators (Gil, 1994), making successful plans available to the learner (Wang, 1995; Yang et al., 2007; Zhuo et al., 2010; Cresswell and Gregory, 2011), excluding action failures (Amir and Chang, 2008), or the presence of a teacher (Benson, 1996). Such knowledge is unlikely to be available to an autonomous agent learning the dynamics of its domain.

Furthermore, only a few relational approaches tackle learning under partial observability (Amir and Chang, 2008; Yang et al., 2007; Zhuo et al., 2010), or noise in any form (Benson, 1996; Pasula et al., 2007; Rodrigues et al., 2010a). The types of partial observability and noise may vary. Observations may be partially observable or noisy in that some fluents are respectively missing (Amir and Chang, 2008) or altered (Benson, 1996; Rodrigues et al., 2010a). Alternatively, sequences of observations may be partially observable in that entire observations may be missing, possibly in combination with some missing fluents within observations (Yang et al., 2007; Zhuo et al., 2010). Finally, observations may be fully observable and noiseless but actions with noisy outcomes may be modelled (Pasula et al., 2007). Of particular note is that no relational approaches currently attempt to learn action models under both partial observability and noise. Similarly, very few approaches support probabilistic operators (Benson, 1996; Pasula et al., 2007; Safaei and Sani, 2007).

The expressiveness of rules which can be learnt also varies, ranging from STRIPS (Halbritter and Geibel, 2007), possibly extended by conditional effects (Wang, 1995; Zhuo et al., 2010; Rodrigues et al., 2010a), to rules which include quantifiers and logical implications (Zhuo et al., 2010), or noisy deictic rules (Pasula et al., 2007). Expressivity not only affects the kinds of domains which may be learnt, but also the tractability of the approach (see Section 2.3). The learnt rules themselves are usually explicitly stated, but need not be; instead the rules may be implicit in a model which predicts the outcome of an action given an initial state (Halbritter and Geibel, 2007; Xu and Laird, 2010). Another distinction is whether methods use online, incremental approaches or learn in an offline manner. Most of the above cited approaches are online and incremental but there are exceptions (Benson, 1996; Pasula et al., 2007; Yang et al., 2007; Zhuo et al., 2010; Cresswell and Gregory, 2011).

In summary, in terms of learning action models representing the affordances available to an autonomous agent, there are a number of requirements which existing approaches do not meet in full. Suitable methods must not assume domain information beyond the grounded symbols which could be expected to be learnt by the methods discussed in Section 2.5.1. Methods must also work in partially observable, noisy domains, which only Pasula et al. (2007) and Rodrigues et al. (2010a) handle to any extent, and neither completely. Additionally, methods should learn in an online, incremental fashion in order to learn from data generated by an agent exploring its world.

As mentioned above, although the MDP framework supports learning of probabilistic transition functions, the existing methods do not support relational domains. Most methods which use alternative abstractions to learn in relational domains do not support probabilistic operators, with a few exceptions. Therefore in this thesis I also focus on learning deterministic operators, although I discuss how the process might be extended to probabilistic operators in Section 6.5.3.

2.6 Deictic reference

The notion of *deictic reference* is central to this thesis, underpinning both the state representation used and the learning process. The term deixis originates from Ancient Greek meaning “pointing” or “showing”. In language, deixis is the term for expressions which refer to the context of an utterance, for example, “this” or “here” (Finegan, 1998). The context must be known for the expression to be correctly understood. Deictic expressions can operate on a spatial (“here”), temporal (“now”) or personal (“I”) basis (among others). In particular, when used for spatial reference, deixis may require a pointing gesture to distinguish the referent, and in general, deixis corresponds to a pointing relation between a word and a context. The term *indexicality* is often used interchangeably with deixis, although in the philosophical tradition indexicality includes deixis but is more general, covering any pointing or indicating relation, for instance, smoke as an indicator or index of fire, or a sundial as an indicator of the time of day (Peirce, 1931).

The idea of deixis has been adopted in the cognitive robotics field, where a deictic reference is a pointer to objects which have a particular role in the world, with object roles coded relative to the agent or current action. Agre and Chapman (1987) introduced deictic references in the form of "indexical-functional entities" which refer to objects in the game of Pengo in terms relative to the main agent in the game, e.g., "the-block-I'm-pushing" or "the-bee-on-the-other-side-of-this-block-next-to-me". Deictic references are only maintained for objects close to the agent, so other objects are effectively invisible to it. As the state changes, objects may move in or out of range, and change role. Each deictic reference must therefore be kept up-to-date so that it always points to an object with the same role.

In the context of reinforcement learning, Whitehead and Ballard (1991) use deictic references (*markers*) to point to target objects in the world. Their system uses two types of marker, *overt* markers which are associated with overt actions that change world state, and *perceptual* markers which are used to gather additional perceptual information, and have associated perceptual actions which can change the object the marker is bound to. Since the deictic representation entails incomplete state observations, to successfully learn a policy their system must additionally learn where to direct its perceptual marker, in order to observe parts of the world state which distinguish between otherwise identical observations.[8]

Deictic references have also been used to recode a first-order description of the world in terms of the arguments of the current action (Benson, 1996; Pasula et al., 2007). Encoding a state using deictic reference involves assigning *deictic terms* to objects in the world. Each deictic term uniquely defines an object or set of objects in the current context. In the approach taken by Benson (1996) and Pasula et al. (2007), each domain constant and each argument of the action is considered to be a deictic term in its own right (referring to the actual object which is the argument for the specific action instance, similar to Whitehead and Ballard's overt markers). Then any object can be assigned a deictic term if it is (positively) related only to objects which have themselves already been assigned deictic terms. The new deictic term for the object is written in terms of the existing deictic terms and the relationships with the object. The

[8]The problem of perceptual aliasing.

process is repeated until no further objects can be coded via deictic reference.

For example, consider a BlocksWorld-type action to pick up block `X` `(PICKUP X)`, where block `X` is on block `Y` `(ON X Y)` and `Y` is on block `Z` `(ON Y Z)`. Writing the deictic terms in the form ⟨deictic term label⟩:⟨constraints⟩, the deictic terms would be:

- $\texttt{arg}_1$`:(PICKUP `$\texttt{arg}_1$`)` [the-block-I'm-picking-up]
- $\texttt{dref}_1$`:(ON arg1 `$\texttt{dref}_1$`)` [the-block-under-the-block-I'm-picking-up]
- $\texttt{dref}_2$`:(ON `$\texttt{dref}_1$` `$\texttt{dref}_2$`)` [the-block-under-the-block-under-the-block-I'm-picking-up]

It is possible for a set of objects to have the same deictic term. Where more than one object shares the same deictic term, Pasula et al. (2007) do not give a deictic term to any of the objects in the set, while Benson (1996) does not specify how this situation should be handled.

There are a number of benefits in using a deictic representation. It reduces the size of the state representation, by limiting the observations to a small number of objects. Providing the roles corresponding to the deictic references are relevant to the task, the reduced representation makes learning of domain dynamics easier, as there are fewer possible states and actions to consider. The relevant deictic terms may be themselves learnt (Whitehead and Ballard, 1991) or be obtained by creating deictic terms for all possible relevant roles (Benson, 1996; Pasula et al., 2007), although this leads to a corresponding increase in the size of the state space. Additionally, deictic references permit 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.

How might a deictic representation arise? Any agent needs to be able to connect its actions with the world. This is particularly apparent in situated cognition, where it has been proposed that the external world could be used as an external visual memory (Brooks, 1990; O'Regan, 1992), reducing (or in the extreme, removing) the need for internal representation of the world. If the world is acting as a memory, this implies a need for a mechanism to address this memory, so that parts of the visual scene can be returned to when required.

A possible mechanism is given by the visual indexing hypothesis (Pylyshyn, 2000), which proposes that humans can maintain pointers to objects in the world (a visual index or "FINger of INSTantiation" known as a FINST (Pylyshyn, 1989)). The main purpose of a visual index is to bind an argument of a mental relational predicate or motor command to a real-world object. Thus, a visual index differs from putative object representations such as object files (Kahneman et al., 1992), event files (Hommel,

2004) or object-action complexes (OACs) (Krüger et al., 2011), in that these are proposed as temporary instantiations of the collection of features and affordances of an object, whereas the visual index is the link between them and the object to which they refer. Pylyshyn (1989) suggests that visual indices may be assigned by either bottom-up or top-down processes, where the bottom-up approach operates at a preconceptual level, on entities which the visual system rapidly identifies as possible objects (proto-objects), in contrast to the top-down approach which may attach indices to conceptual objects identified by their features. Experimental evidence suggests the availability of a small number of visual indices, between 4 and 10 (Pylyshyn, 2000; Bullot and Droulez, 2008), which point to (proto-)objects rather than locations (Kahneman et al., 1992; Pylyshyn, 2000), and track those (proto-)objects over time (Pylyshyn, 2000).

There are clear parallels between visual indexing, deictic reference and attention. A visual index is clearly a deictic reference: it points from the representation of an object with a particular role (the argument of a mental relational predicate or motor command) to an object in the world. Ballard et al. (1997) describe attention as a "neural deictic device", while Hurford (2003) explicitly identifies deictic reference with attention, although Pylyshyn (2001, 2009) argues slightly differently that visual indexing is one stage in a series of processes which constitute attention.

Moreover, Hurford (2003) proposes that there are neural correlates of the full predicate-argument structure of a logical formula, where deictic references, in some form, take the role of arguments in relational predicates. He relates the differential processing of sensory inputs in the ventral and dorsal streams (Goodale and Milner, 1992) to the respective construction of predicates and arguments. The dorsal stream, concerned with visuomotor control and the capture of egocentric locational properties of objects corresponds to the maintenance of deictic references and thence arguments, while the ventral stream, concerned with perception of object properties, corresponds to the creation and maintenance of predicates.

Deictic representations therefore support the goal of grounding representations in the real world while also enabling the creation and maintenance of predicate-argument structures necessary for reasoning in relational domains. Since deictic coding may come about through attentional mechanisms already well-explored in the literature (e.g., saliency maps (Koch and Ullman, 1985; Itti et al., 1998)), the work in this thesis will assume some form of deictic coding mechanism.

Furthermore, although not the main motivation for choosing a deictic representation, visual indexing provides solutions to both the correspondence problem and the binding problem discussed in Section 2.2.1. By providing an object tracking mechanism across different observations, object correspondence is maintained, while feature binding is achieved since every feature of a specific object can be associated with a particular visual index. As a result, a solution to the binding and correspondence problems will be assumed in the remainder of this thesis.

2.7 Summary

In this thesis I aim to induce action representations based on the actions of an agent in the world, and grounded in the affordances the agent perceives. The resulting representation is intended to support a planner by providing STRIPS-like rules encoding the relationship between the state of the external world and the actions it affords, and between affordances and the states resulting from the affordances. Previous work on learning grounded action models has demonstrated that object, predicate and action symbols (including action parameters) can be grounded in sensorimotor representations. Although there has been work on learning relational action models, none of the approaches so far are suitable for learning in partially observable, noisy environments typical of robot domains.

In setting up an algorithm to learn action models, the choice of representation is important as it determines the set of domains which a method may be applied to, while increasing expressiveness is also associated with decreasing computational efficiency, to the extreme of intractability. Finally, deictic references generate state representations which are useful for action model learning and have been successfully employed in previous work.

Chapter 3

Preliminaries

In this chapter I set out the formal definition of domains and states which will be used in this thesis. I also define a form of object equivalence based on deictic reference, which will underlie the learning models used in later chapters. Additionally I discuss the use of simulated data generated from relational planning domains, which domains are used, the process of generating the data, and how partial observability and noise are modelled in this context.

3.1 Definitions

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* is a statement of the form:

(i) $p(c_1, c_2, \ldots, c_n)$, where $p \in \mathcal{P}$, n is the arity of p, and each $c_i \in \mathcal{O}$, or

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

A *state* is any set of fluent expressions, and $\mathcal{S}$ is the set of all possible states. For any state $s \in \mathcal{S}$, a fluent expression ϕ is true at s iff $\phi \in s$. The negation of a fluent expression, $\neg\phi$, is true at s (also, ϕ is false at s) iff $\phi \notin s$.[1] 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 fluent expressions and negated fluent expressions.

I consider two kinds of action effects. Firstly, standard STRIPS effects, where each

[1]One of the properties of functions is that they have unique mappings. Thus, the specification of fluent expressions over the same function, but with different mappings, could give rise to inconsistent states (e.g., a state where $f(a) = c$ and $f(a) = d$ both held). However, the representations used in Chapters 4 and 5 do not support this possibility.

$e \in Eff_a$ has the form $add(\phi)$ or $del(\phi)$, and ϕ is any fluent expression. Secondly, *conditional effects* of the form $C_e \Rightarrow add(\phi)$ or $C_e \Rightarrow del(\phi)$. Here, C_e is any set of fluent expressions and negated 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 O is said to be an *action instance*. For any fluent expression or action ϕ, the function $args(\phi)$ returns the set of arguments of ϕ, $\{c_1, c_2, \ldots, c_n\}$, and the function $args_i(\phi)$ returns the i-th argument of ϕ. For any fluent expression or action ϕ, the function $label(\phi)$ returns its predicate or action symbol.

3.2 Using deictic references to shape the rule space

In Chapters 4 and 5, I will discuss a model for learning action rules — effects and preconditions of actions — given examples of actions and their prior and successor states. The problem of learning rules is formulated as a classification problem, where each classifier takes as input a state and an action, and outputs whether some substate changes or not. Target outputs used for training will be in the form of deltas, the difference between each prior and successor state.

There are three related issues with this approach which will be handled by using deictic references. Firstly, the space of possible preconditions and effects is exponentially large, as potentially any combination of fluents could constitute a precondition or effect. As discussed in Chapter 2, most approaches to the problem of learning action rules therefore restrict the space in some way; similarly, action languages used for planning restrict the possible form of the rules. The approach here will also be to constrain the rule space.

The specific restriction on the space of rules is based on intuition about how actions affect the world. Usually, an action only affects some small part of the world, and the objects which are affected typically have some connection to the action, or to objects involved in the action. For example, in a world of stacking blocks, stacking one block on top of another changes the properties of the stacked block, and the block being stacked upon, but not the properties of other blocks lying on the table. Often the affected objects are parameters of the action, or are directly related to the action parameters, as in the example above. Sometimes the affected objects may be indirectly related via relationships with intermediary objects, such as if the bottom block in a stack of several blocks is moved, causing all the blocks in the stack to fall on to the table. Generally, though, objects are unaffected by an action if they have no relations with the parameters of the action. Likewise, action outcomes are unaffected by the state of objects which have no relations with the action parameters. The rule space can therefore be reduced by ignoring objects which are unrelated to the action parameters, essentially the same approach as that taken by Benson (1996) and Pasula et al. (2007).

A second issue is that if two different objects have the same role in an action, the learning model should treat those objects in the same way. For instance, if a BlocksWorld `STACK` action is to stack `Block3` on `Block5` with the result `(ON Block3 Block5)` then we expect if we replace `Block3` and `Block5` with `Block8` and `Block1` that `(STACK Block8 Block1)` should result in `(ON Block8 Block1)`: the blocks' labels have no significance. This is clearly desirable when the objects are in the same position in the action parameters of different instances of an action: if the states are identical in structure, we expect that the objects will be affected by the action in the same way. The principle also extends to objects which are not action parameters. If objects in different states have exactly the same set of relations with the action parameters, we would expect that those objects would affect, and be affected by, the action in the same way. Finally, if two objects in the *same* state have the same role (which cannot be a shared position in the action parameter list, so must be an identical set of relations with the action parameters), then we also expect that those objects will affect, and be affected by, the action in the same way. One approach to handling this is for the learning model not to use constants to represent individual objects, but instead use a representation which accounts for object roles, a requirement which again may be fulfilled by employing a deictic representation (Benson, 1996; Pasula et al., 2007).

Lastly, a similarity measure between states is needed to support the use of a similarity-based classifier. States or deltas are not usually directly comparable since the specific objects involved vary, so simply calculating a difference between states, as used to construct deltas, is inadequate. One case is trivial: a comparison can be made if the states are restricted so that each state only includes objects which are parameters of the current action, as then a one-to-one mapping between objects in the same parameter positions can be constructed. The issue of measuring the similarity between states has mostly not arisen in prior work since either the learning models did not use similarity-based classification, or the deltas were comparable because only the action parameters were considered (Halbritter and Geibel, 2007). Xu and Laird (2010) do present a form of similarity matching in this context, but their heuristic aims only to find the previously seen state which is most structurally similar to the current state, rather than giving states a similarity score.

Underlying all of these issues is the idea that objects should be represented by their role in an action, where the role of an object depends on how it is related to the action and the rest of the state. An object may be directly or indirectly related to an action, where these are defined recursively as follows.

Definition 3.2.1. An object is *directly* related to an action if it is a parameter of the action. An object is *indirectly* related to an action if it has a relation with another object which is itself directly or indirectly related to the action.

Formally, I make the following assumptions:

Relatedness Assumption: An object's state can only affect or be affected by an action if the object is related to the action, either directly or indirectly.

Object Role Assumption: An object's role in an action is determined by its properties, its relation to the action, and its relations with other objects in the current state, relative to the action. That is, an object's role is determined by its deictic reference, relative to the action.

The Relatedness Assumption seems intuitive: we would not expect an action to change something in the real world unless there was some sort of relation between the things being acted upon and the things which changed. If we do not perceive or have knowledge of a suitable relation we would probably posit the existence of one. Since such predicate invention is beyond the scope of this thesis, it will be assumed that the set of relations given in any state description is complete. As a result the set of objects which have deictic references relative to the action are assumed to contain the objects relevant to the preconditions or effects of an action, implicitly assumed in the work of Benson (1996) and Pasula et al. (2007).

The Object Role Assumption is similar to the assumption of *Sufficiency of Object Properties* made by Gardiol and Kaelbling (2007) where the function of an object is assumed to be entirely determined by its attributes and relations with other objects in the world, except that here the relation between the object and the current action is included. The purpose of the Sufficiency of Object Properties assumption was to support the definition of an equivalence relation on actions, via the idea that objects are equivalent if they have the same attributes and relations with other objects in the world, and that states are equivalent if every object in one state has an equivalent object in the other state and vice versa. For instance, a BlocksWorld state where `Block1` and

`Block2` are on the table, and `Block3` is on `Block1` is equivalent to a state where `Block4` and `Block5` are on the table, and `Block6` is on `Block5`, exactly *because* each block in one state has an equivalent in the other state.

The Object Role assumption has two purposes. Firstly, it means that objects can be represented in state descriptions and action rules by their deictic terms, instead of their labels, forcing any model of state dynamics to treat objects with the same role in the same way (as also in Benson (1996); Pasula et al. (2007)). One difference in implementation in this work is that any objects which share a deictic term are represented by that single deictic term, because those objects are indistinguishable to the action.

Secondly, the Object Role assumption serves a similar purpose to the Sufficiency of Object Properties assumption in that it supports the definition of a similarity relation between states via an equivalence relation on objects. By considering the roles of the objects in two states or deltas, it is possible to map objects in one state or delta to another. It can then be checked which relations hold in both states or deltas, to give a measure of similarity between them.

Below I first give a formal definition of deictic terms (following Pasula et al. (2007)), followed by a definition of the equivalence of objects discussed above.

Definition 3.2.2. A *deictic term* is a variable v_i and an associated constraint ρ_i where ρ_i either:

(i) is an equality relation with one of the arguments of the current action, or

(ii) is a set of literals defining v_i in terms of other variables v_j, where $j < i$, and at least one literal is positive.[2]

(Since a deictic term may describe more than one object, I will also refer to the set $\{v : \rho_i\}$ as a deictic term.)

If an object x satisfies ρ_i then x has the deictic term $v_i : \rho_i$. The set of all deictic terms of x is denoted $dterms(x)$.

Definition 3.2.3. Two objects x and y (in the same or different states with associated actions) are equivalent ($x \sim y$) iff $dterms(x) = dterms(y)$.

[2]The requirement for at least one positive literal is due to the open world assumption used in this thesis (see Section 5.1.1.1), in contrast to the closed world assumption made by Pasula et al. (2007) and Benson (1996).

The notion of equivalence can be extended to fluents as follows:

Definition 3.2.4. Relations r and r' (in the same or different states) are equivalent ($r \sim r'$) iff:

- $label(r) = label(r')$, and
- $args_j(r) \sim args_j(r'), \forall j$.

The equivalence relations in both definitions underlie the learning models used in Chapters 4 and 5.

3.3 Data

The learning model will mostly be tested on simulated data, because of the difficulty of obtaining relational observations from robots. As discussed in Section 2.5, autonomous learning of relations is an unsolved problem. The alternative, using predefined rules to determine relationships, is fragile, as it is not always clear-cut when a relation holds, nor do predefined rules necessarily coincide with the context required for a particular action. Therefore, as in much previous work in learning relational action models (Pasula et al., 2007; Yang et al., 2007; Amir and Chang, 2008), I use simulated data generated from relational planning domains.

The planning domains I use are taken from various incarnations of the International Planning Competition (IPC).[3] The domains are described in PDDL (McDermott et al., 1998), the standard representation language of the IPC. An example PDDL description of the BlocksWorld domain is shown in Figure 3.1 (PDDL descriptions of other domains used can be found in the Appendix). The domains differ in terms of their complexity in several different dimensions: the number and arity of actions, predicates and functions; the number and hierarchy of types; conditional effects; and preconditions containing implications. The main characteristics of the domains are detailed in Table 3.1.

The simplest domains in this set are the pure STRIPS domains (BlocksWorld, Depots, ZenoTravel and DriverLog). The rules describing the action models of these domains only refer to objects listed in the parameters of the actions. There are no negative preconditions, no quantifiers and no conditional effects. The effects and preconditions are simple conjunctions of fluents. Although classed as a STRIPS domain, several of

[3] `http://ipc.icaps-conference.org/`

```
(define (domain blocksworld)
  (:requirements :strips)
  (:predicates (arm-empty)
    (clear ?x)
    (ontable ?x)
    (holding ?x)
    (on ?x ?y))

(:action pickup
  :parameters (?ob)
  :precondition  (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob)) (not (arm-empty))))

(:action putdown
  :parameters (?ob)
  :precondition (holding ?ob)
  :effect ((and (clear ?ob) (arm-empty) (on-table ?ob) (not (holding ?ob))))

(:action stack
  :parameters (?ob ?underob)
  :precondition (and (clear ?underob) (holding ?ob))
  :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob) (not (clear ?underob))
          (not (holding ?ob))))

(:action unstack
  :parameters (?ob ?underob)
  :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
  :effect (and (holding ?ob) (clear ?underob) (not (on ?ob ?underob)) (not (clear ?ob))
          (not (arm-empty)))))
```

Figure 3.1: PDDL description of the BlocksWorld domain.

the actions in the Rovers domain are unusual, in that some effects are not changes. Instead these actions always (unconditionally) set a fluent p to true, regardless of p's previous value. In a system which learns effects as changes (as in this thesis), rather than as values, the action behaves differently depending on whether p is false or true (either p changes or does not). Such behaviour *can* be encoded by setting up the effect as a conditional effect, e.g.,

```
(:action take_image
   :parameters (?r - rover ?p - waypoint ?o - objective ?i - camera ?m - mode)
   :precondition (and (calibrated ?i ?r) (on_board ?i ?r) (equipped_for_imaging ?r)
                      (supports ?i ?m) (visible_from ?o ?p) (at ?r ?p))
   :effect (and (have_image ?r ?o ?m)(not (calibrated ?i ?r))))
```

becomes

```
(:action take_image
   :parameters (?r - rover ?p - waypoint ?o - objective ?i - camera ?m - mode)
   :precondition (and (calibrated ?i ?r) (on_board ?i ?r) (equipped_for_imaging ?r)
                      (supports ?i ?m) (visible_from ?o ?p) (at ?r ?p))
   :effect (and (not (calibrated ?i ?r))
                (when (not (have_image ?r ?o ?m))
                           (have_image ?r ?o ?m))))
```

Domain	Actions		Predicates				
	No.	Max arity	No. (+types)	Max arity	STRIPS	Conditional effects	Negative preconditions
BlocksWorld	4	2	5	2	Y		
Depots	5	4	6 (+6)	2	Y		
ZenoTravel	5	6	8 (+4)	2	Y		
DriverLog	6	4	6 (+4)	2	Y		
Briefcase	3	2	3 (+2)	2		Y	Y
Elevator	3	2	6 (+2)	2		Y	Y
Rovers	9	6	25 (+7)	3	Y	Y	

Table 3.1: Domain characteristics

It is therefore this type of conditional rule which the models in later chapters must learn for the Rovers domain, and so in Table 3.1, Rovers is listed as a conditional effects domain. The more complex ADL domains (Briefcase and Elevator) have more complex rules. Both have negative preconditions, conditional effects, and use universal quantification.

3.4 Data generation

To simulate an agent "motor-babbling" in the world, sequences of random actions and resulting states were generated from PDDL domain descriptions and used as training and testing data. All data was generated using a script, based on the Random Action Generator 0.5 available at `http://magma.cs.uiuc.edu/filter/`. The script requires a PDDL domain description and an initial state as input. The initial states were generated at random by the BlocksWorld state generator (Slaney and Thiébaux, 2001); for ZenoTravel, Depots, DriverLog and Rovers[4] by the IPC3 problem generator (`http://planning.cis.strath.ac.uk/competition/domains.html`); and for Briefcase and Elevator (Miconic-SIMPLE domain) by the FF domain collection generators (`http://www.loria.fr/~hoffmanj/ff-domains.html`). The number of objects in the state space was higher in the test data than in the training data, to demonstrate that the learnt models could be applied across different instances of the same domain. The

[4]The IPC3 Rovers domain description contains several actions whose effects list contains both an addition and deletion of the same fluent. This encoding is intended to restrict possible plans in the domain, and is not relevant to the action dynamics. Therefore, for the purposes of learning an action model, these superfluous effects are ignored.

Domain	**Training**	**Testing**
BlocksWorld	13 blocks	30 blocks
Depots	1 depot	4 depots
	2 distributors	4 distributors
	2 trucks	4 trucks
	3 pallets	10 pallets
	3 hoists	8 hoists
	10 crates	8 crates
ZenoTravel	5 cities	10 cities
	3 planes	5 planes
	7 people	10 people
DriverLog	3 road junctions	20 road junctions
	3 drivers	5 drivers
	7 packages	25 packages
	3 trucks	5 trucks
Briefcase	50 objects	100 objects
	50 locations	100 locations
Elevator	15 floors	30 floors
	5 passengers	20 passengers
Rovers	2 rovers	4 rovers
	4 waypoints	8 waypoints
	3 objectives	4 objectives
	3 cameras	4 cameras
	3 modes	3 modes
	2 stores	4 stores
	1 lander	1 lander

Table 3.2: Numbers of each type of object in training and testing worlds

numbers of each type of object in training and testing worlds for each domain is shown in Table 3.2.

To determine an error bound on the results, 10 runs with different randomly generated training and testing sets were used. Each training set was a sequence of 20000 actions, and each testing set a sequence of 2000 actions. Both sequences contained an equal mixture of successful and unsuccessful actions (where some precondition of the action was not satisfied, and so no change occurred in the world).

In some domains, portions of the state space can only be traversed once. For example, in the Elevator domain, passengers are delivered by the elevator to their destinations, and never re-enter the elevator. Once all the passengers have been delivered, the only actions which can be performed are for the lift to travel up and down between floors, and so the opportunities for learning the action model are much reduced.

In these domains (Elevator, Rovers), multiple shorter sequences of 400 actions were generated from randomly generated starting states.

3.5 Models of noise and partial observability

In the real world, an agent's observations of the world state are subject to partial observability and noise. Partial observability can arise due to sensor failures (e.g., an unreliable sensor sometimes does not give a reading for a particular property of the world) or due to sensor limitations (e.g., usually an agent cannot detect objects which are physically located elsewhere). Similarly, noise originates from noisy sensors and effectors. If probabilistic outcomes are not being considered, world non-determinism can also contribute to noise in the observations, since, like effector noise, it leads to different outcomes from situations which are the same.

An agent interacts with the world by making observations and performing actions. The action learning problem is to predict the new world state given the current world state and an action. Formally, at time t, an agent observes the world state $\bar{s}_t$ through its (possibly noisy, unreliable) sensors, producing an observation $\bar{z}_t$. The raw sensor data may be further refined into a set of percepts $\bar{p}_t$ via a *perceptual function* f (Modayil and Kuipers, 2008). The subsequent world state, $\bar{s}_{t+1}$, is determined (possibly stochastically) by the previous state, $\bar{s}_t$, and whichever action a_t the agent chooses to perform (using possibly noisy effectors). The resulting sequence of percepts and actions forms a set of training examples where each $\langle \bar{p}_i, a_i \rangle$ is an input with corresponding target $\bar{p}_{i+1}$. The examples can then be used to train a learning model to predict the new world state, as observed by the agent, i.e., to predict $\bar{p}_{t+1}$, given the current observation of world state, $\bar{p}_t$, and an action a_t (Figure 3.2).

This *world-level observation model* holds for agents operating in the real-world, but often the world and the agent must be simulated. One approach is to build a physical model of the agent and the world (Uğur and Şahin, 2010), for example, by using a physics engine such as ODE (`http://www.ode.org`). Alternatively, data can be generated under the assumption that all behaviour is fully-observable and deterministic, and then modified. A popular approach is to insert a *blocking process* (Schuurmans and Greiner, 1997) which degrades the observations from a fully-observable, deterministic world. For example, consider the case where the sensor data is a vector of Boolean values $\bar{z} = \langle z_1, z_2, \ldots, z_n \rangle$, $z_i \in \{0, 1\}$ and the set of possible observed states of the world is $Z = \{0, 1\}^n$. A blocking process models partial observability by mapping each z_i to

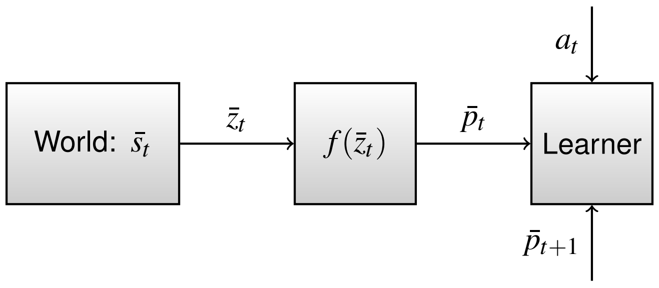


Figure 3.2: *World-level observation model* generating training data for the learner from world state observations. At time t, the agent makes an observation $\bar{z}_t$ which is converted into percepts $\bar{p}_t$ via some function f. The training data supplied to the learner consists of inputs $\langle \bar{p}_t, a_t \rangle$ and targets $\bar{p}_{t+1}$.

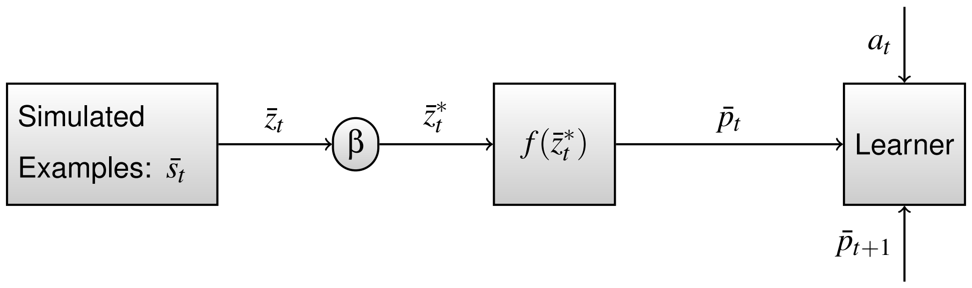


Figure 3.3: *World-level blocked-observation model* where observations are modelled using a blocking process β on sensor readings and then percepts are derived from the modified sensor values.

an unknown value $*$ with some probability. A trivial extension is to model noise by mapping each z_i to $(1 - z_i)$ with some probability. The blocking process approach is shown in Figure 3.3.

However, it can be difficult to define the perceptual function f which maps sensor values to percepts. A domain expert can predefine the mapping, but the result is a domain-specific mapping, which is likely to be inflexible in the face of unexpected situations. Alternatively, the mapping to percepts may be learnt. Research in the autonomous robotics field has focused on learning non-relational percepts, such as learning to identify objects and properties of objects (Modayil and Kuipers, 2008). Little progress has been made in acquiring relational percepts, apart from recent work by Rosman and Ramamoorthy (2011) on learning spatial relations between objects, specifically, “on” and “is adjacent to”. As a result, generating observations of rela-

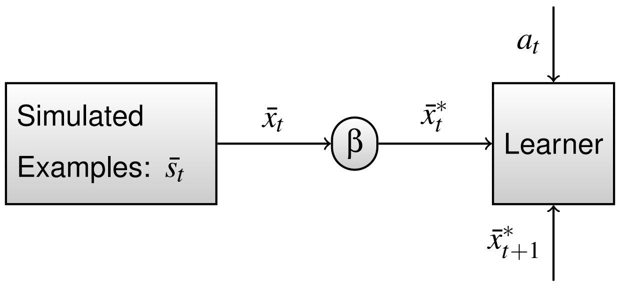


Figure 3.4: *Percept-level blocked-observation model*) which models observations using a blocking process β on examples which are simulated percepts.

tional data requires either a predefined mapping from sensory data to percepts, or the use of simulated relational data. Therefore the model of the interaction between an agent and the world is often further simplified (Figure 3.4). The world is modelled in terms of percepts $\bar{x}_t$, and then a blocking process is applied to the set of generated percepts at each time t (Yang et al., 2007; Amir and Chang, 2008; Bouthinon et al., 2009; Rodrigues et al., 2010a).

Applying a blocking process (with noise) to observations, without accounting for dependencies between components of the observation, whether sensor-based or percept-based, can have the undesirable effect of generating inconsistent observations of the world. For example, in BlocksWorld, ⟨`(on A B)`,`(clear A)`,`(on-table B)`⟩ could be degraded to ⟨`(on A B)`,`(holding A)`,`(clear A)`,`(on-table B)`⟩, where block A is now both on block B and in the gripper. Applying noise and then somehow making the observations consistent is more complex, since it requires knowledge of how the relations in the world affect each other. Alternatively, observations without noise can be used, with a randomly chosen, noiseless state used as the outcome, to model the situation where the prior observed state was consistent, but incorrect, and therefore the action had an unexpected effect. All of these approaches to modelling noise give rise to different forms of perceptual aliasing, where two distinct states cannot be differentiated because their observed descriptions are the same.

Few reports in the literature relate to the application of noise to simulated sensor data or percepts. The blocking process approach is used by Rodrigues et al. (2010a) to apply random noise to percepts. Although not explicitly modelling noisy observations, the approach in Pasula et al. (2007) models some outcomes as noise outcomes, which could also cover unexpected outcomes resulting from perceptual aliasing.

Chapter 4

Learning STRIPS action models

The work presented in this chapter tackles the problem of learning action models in noisy, partially observable STRIPS domains, which may additionally have conditional effects. The contribution of the chapter is to demonstrate that decomposing the states via deictic reference is a viable approach, which can then be generalised to learning action models in more challenging domains. Below I describe the model's state representation, and how it is constructed using deictic references. I present a standard learning model which can learn action models using this representation, and discuss results from applying the approach in different planning domains.

4.1 Strategy

The task of the learning mechanism is to learn the associations between action-precondition pairs and their effects, that is, rules of the form $\langle A, \mathit{Pre}_A \rangle \rightarrow \mathit{Eff}_A$. My approach will be to encode the learning problem in terms of the inputs and outputs of a set of classifiers: change to a single fluent for a particular action can be predicted by a single classifier, taking as input a state description. The full set of changes to a state as a result of an action can then be constructed by combining (by conjunction) the changes predicted by each classifier. Note that whenever a classifier predicts that a fluent will not change, it means that the preconditions for an action to change that fluent are not satisfied. Thus, provided that action failures as well as successes are input to the classifiers, preconditions as well as effects of actions can be learnt using this approach. This learning strategy has implications for the types of action rules which can be learnt. Since action effects are constructed by combining changes to individual fluents by conjunction, only conjunctive effects can be learnt. Conditional effects, where in some situations there are extra effects of an action, can also be learnt, since the conditions for change to each individual fluent are learnt separately.

However, predicting change for every possible fluent in a world can be computationally expensive, since for a world with n objects and with predicates whose maximum arity is m, the number of fluents to be predicted is $O(\binom{n}{m})$. The same problem affects the representation of the input states, since I will remove the closed world assumption in order to differentiate between unobserved and false fluents when working in partially observable domains.[1] The first step towards representing the state for the learning model is therefore to identify a set of objects to include, or attend to, in a reduced state description. The aim is to reduce the number of represented objects in order to make the learning problem more tractable, ideally without removing objects which are relevant to the action.

The objects to include in the state description are identified via deictic references, relative to the current action. The deictic references can be restricted to include more or fewer objects, depending on how distantly related the objects are to the action parameters. The smallest set of objects would just be the set of action parameters; the largest set would include the action parameters and any object which has a (true) relation with an object already known to be in the set (i.e. the full transitive closure). Here I use the simplest case where the state description only includes objects listed in the action parameters. Note that, by the definition of STRIPS actions, this is sufficient for learning action models in STRIPS domains, because of the STRIPS scope assumption (Section 2.2.2) that all objects which appear in the preconditions or effects of an action also appear in its argument list. In Chapter 5, deictic references beyond action parameters are considered.

Figure 4.1a presents an example from the BlocksWorld domain, in which an agent can manipulate a set of blocks on a table. Given the action `(stack A B)`, to stack block A on top of block B, the action parameters are $\{A, B\}$. Each parameter has a deictic term based on its position in the action parameter list, e.g. $arg1$ is the deictic term for A, and $arg2$ for B. The remaining objects are omitted from the learner's state description.

[1]A possible alternative to removing the closed world assumption would be to move to a knowledge representation where all fluents are knowledge fluents of the form Kf and $K\neg f$ corresponding to whether the agent has observed f or $\neg f$ to be true. In this case the closed world assumption means that unobserved fluents are assumed to be unobserved. However, since all observed true and false fluents are enumerated, the same problem arises, namely that the number of fluents to represent is large.

4.2 Representation

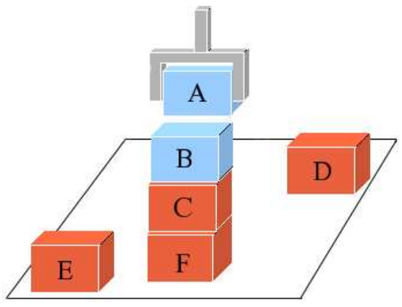
By restricting the state description to only those objects listed in the action parameters, every possible fluent relating the objects can be defined in terms of its label and the positions of its arguments in the action parameter list. As a consequence, states can be represented using an attribute-value representation, where each attribute is a combination of fluent label and argument positions, and each value is the value of that fluent in the state, such as in the vector representation described below.

An input vector, representing an observation of the state space before an action is performed, is constructed as follows. Each action $a \in \mathcal{A}$, and each 0-ary fluent, is represented by a single element of the vector. Each possible fluent relating any object in the action parameters is represented by an element og the vector. The value of an element is 1 (-1) if the corresponding predicate is true (false), or if the corresponding action is (not) the current action. For functions, the value of the element is just the value of the function, if integer-valued, or an ordinal when the function is nominal-valued. Where the fluent represented by an element is unobserved, the element is set to an arbitrary value N. This value is simply a wildcard indicating that the element's true value is unknown, and is not used in any of the perceptron's calculations. Since different actions may take different numbers of parameters, the number of possible fluents varies: vectors are padded with entries set to N up to the maximum possible length of vector for a domain.

An ordering on the fluents in the vector is necessary, so that vectors can be constructed repeatably, and to facilitate similarity comparisons between vectors. The ordering is created by first establishing an ordering on both the predicates, and the objects. The predicates are given an arbitrary ordering $p_1, \ldots, p_m$ which is retained for any example in the domain. The objects are ordered by order of appearance in the action parameter list: $obj_1, \ldots, obj_n$. Obviously the actual objects in a particular position in the parameter list will vary between examples, but the fluents relating to the i-th parameter will always be in the same position in the vector.

For each action parameter obj_i, all the fluents with that parameter as their first argument are arranged in a contiguous block in the vector. The blocks are arranged in the vector in object order. Within each block, all fluents with the same predicate are arranged in a contiguous block, and this set of blocks is arranged in predicate order.

It remains to define an ordering on the remaining arguments of each predicate, to give a full ordering for the vector. Since the first argument is fixed, the other arguments



(a) Example BlocksWorld state

Input vector	Corresponding action/predicate	
-1	`pickup`$(arg1)$	Actions
-1	`putdown`$(arg1)$	
1	`stack`$(arg1,arg2)$	
-1	`unstack`$(arg1,arg2)$	
-1	`armempty`	Object independent properties
	...	
1	`holding`	Properties of $arg1$
-1	`ontable`	
-1	`clear`	
-1	`on`-$arg1$	
-1	`on`-$arg2$	
	...	
-1	`holding`	Properties of $arg2$
-1	`ontable`	
1	`clear`	
-1	`on`-$arg1$	
-1	`on`-$arg2$	
	...	

(b) Input vector

Output vector	Corresponding predicate
1	`armempty`
	...
1	`holding`
-1	`ontable`
1	`clear`
-1	`on`-$arg1$
1	`on`-$arg2$
	...
-1	`holding`
-1	`ontable`
1	`clear`
-1	`on`-$arg1$
-1	`on`-$arg2$
	...

(c) Output vector

Figure 4.1: (a) Action `stack`(A,B) results in objects A and B being attended to, while unrelated objects C, D, E and F are ignored. Objects A and B are referred to by the deictic terms $arg1$ and $arg2$ respectively, in the vector representation shown in (b) and (c).

(b) Input vector representation of the (fully observable) BlocksWorld `stack` action and prior state from (a). The first 4 entries correspond to the 4 domain actions. The entry for `stack` is set to 1 since it is the current action. The 0-ary fluent `armempty` is represented by a single element, set to -1 since the gripper is holding object A. The first set of fluents represented in the vector are those for object A since it is the first parameter of `stack`. The second set of fluents relate to object B, as the second parameter of `stack`.

(c) Output vector representation for the same action and prior state. There are no entries for actions. Elements corresponding to fluents which changed have value 1, and all other elements have value -1.

are drawn from every correctly sized (arity of predicate - 1) subset of the remaining $n-1$ objects. For a ternary predicate p and first argument obj_1, with 10 action parameters the subsets to consider would be $\{obj_2, obj_3\}$,..., $\{obj_2, obj_{10}\}$,$\{obj_3, obj_4\}$,..., $\{obj_3, obj_{10}\}$,..., $\{obj_9, obj_{10}\}$. Note that within each subset there are multiple possible orderings of the objects which could be used to fill the remaining parameters of the predicate. The set of subsets can be ordered lexicographically (e.g. as shown), therefore for each predicate/first-argument combination, fluents whose remaining parameters are contained in the same subset are arranged in contiguous blocks in the vector, with the blocks arranged in lexicographic order.

The ordering of the predicate arguments is important, so obj_i followed by every permutation of each subset is a possible set of arguments to p_j. Using the previous example, this gives arguments (obj_1, obj_2, obj_3), $(obj_1, obj_3, obj_2)\ldots$, (obj_1, obj_2, obj_{10}),$(obj_1, obj_{10}, obj_2)\ldots$, (obj_1, obj_9, obj_{10}), (obj_1, obj_{10}, obj_9). The permutations of each subset can also be ordered lexicographically (as above), and this is the ordering used for the fluents in the vector. If the domain has object types, these can be used to reduce the number of possible fluents, since some types of relation will only take certain types of objects at specific parameter positions.

The form of the output vectors representing an action's effects on a state is identical to the input vectors, except that the actions are excluded from the vector. For predicates, elements are set to 1 if the corresponding predicate changes, and -1 if the corresponding predicate does not change, while elements corresponding to functions are set to the result of subtracting the previous value from the new value. Elements corresponding to unobserved predicates or functions are set to N as in the input vector. Figure 4.1 shows an example of an input and output vector for the `(stack A B)` action in the BlocksWorld domain.

Even with such a highly restricted set of objects, the number of possible fluents in a state description is large. If the maximum number of action parameters is M, then there are $\binom{M(M-1)}{n-1} \times (n-1)!$ entries in the vector for each n-ary predicate. To counter this, it will be assumed that $n < 5$ for any domain. This choice is motivated by research showing that in human cognition, only relations of arity four or lower can be processed directly; higher-arity relations must be decomposed in order to be processed (Halford et al., 1988). Similarly, relations in language typically have a maximum arity of four (Hayes et al. (2001):226, Hurford (2003)). It therefore seems reasonable to assume that, firstly, most domains an agent will encounter will not have relations of arity greater than 4, and, secondly, that if necessary, higher-arity relations can be processed by splitting into many lower-arity relations. Typically, M is also relatively small: for instance, the largest M in the IPC planning domains is 9 in the Woodworking domain (6th IPC). With n and M bounded, the size of the vector representation grows linearly with the number of predicates in a domain.[2]

4.3 Learning

The proposed structure of the learning approach is shown in Figure 4.2. Once the world state is converted into a vector representation, the prediction of each element of the output vector can be treated as a separate supervised learning problem, for which there are a number of potentially suitable classifiers, e.g. SVMs, perceptrons, naive Bayes, neural nets. However, in the context of an autonomous agent learning an action model of its environment, the classifier should ideally be incremental and fast.

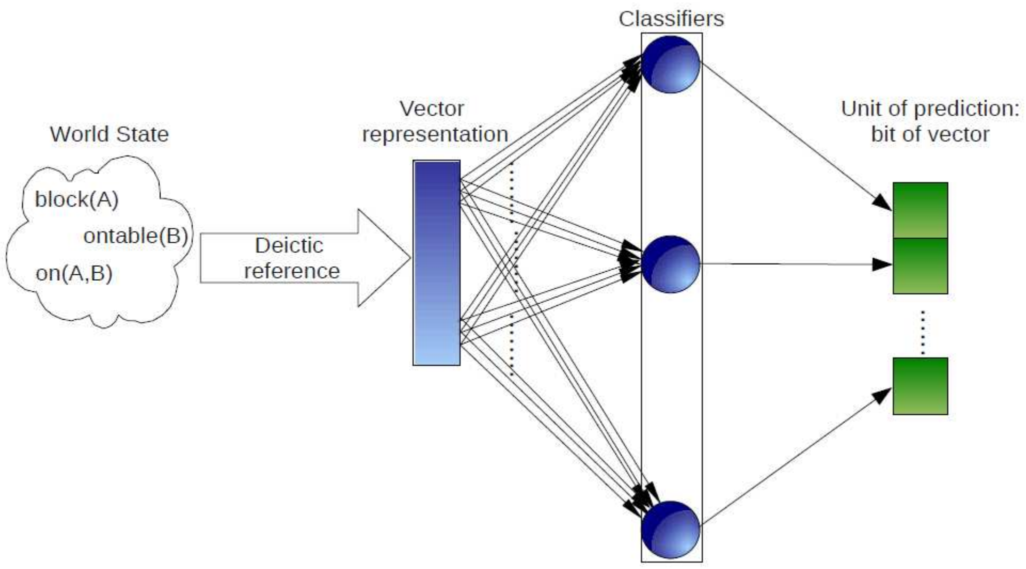


Figure 4.2: Structure of the learning model. World state is first converted into a vector representation. Then each element of the output state vector is learnt/predicted by a different classifier taking as input the current state vector and the action to be performed.

[2]Typically, many of the entries in the vector will be set to N and so it would be more economical to use a sparse representation than the full vector representation discussed here. In particular, the graphical representation discussed in Chapter 5 would meet this requirement.

I therefore use the *perceptron* (Rosenblatt, 1958), a simple yet fast, incremental binary classifier. It maintains a weight vector $\mathbf{w}$ which is adjusted at each training step. The i-th input vector $\mathbf{x}_i \in \mathbb{R}^n$ in a class $y \in \{-1,1\}$ is classified by the perceptron using the decision function $f(\mathbf{x}_i) = sign(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle)$. If $f(\mathbf{x}_i)$ is not the correct class then $\mathbf{w}$ is set to $\mathbf{w} + y\mathbf{x}_i$; if $f(\mathbf{x}_i)$ is correct then $\mathbf{w}$ is left unchanged. By the Perceptron Convergence Theorem, provided the data is linearly separable, the perceptron algorithm is guaranteed to converge on a solution in a finite number of steps (Block, 1962; Novikoff, 1963; Minsky and Papert, 1969). If the data is not linearly separable then the algorithm oscillates, changing $\mathbf{w}$ at each misclassified input vector.

One solution for non-linearly separable data is to map the input feature space into a higher-dimensional space where the data is linearly separable. However, an explicit mapping may lead to a massive expansion in the number of features, making the classification problem computationally infeasible. Instead, an implicit mapping is achieved by applying the *kernel trick* to the perceptron algorithm (Freund and Schapire, 1999), by noting that the decision function can be written in terms of the dot product of the input vectors:

$$f(\mathbf{x}_i) = sign(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle) = sign(\sum_{j=1}^{n} \alpha_j y_j \langle \mathbf{x}_j \cdot \mathbf{x}_i \rangle),$$

where α_j is the number of times the j-th example has been misclassified by the perceptron. By replacing the dot product with a *kernel function* $k(\mathbf{x}_i, \mathbf{x}_j)$ which calculates $\langle \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) \rangle$ for some mapping ϕ, the perceptron algorithm can be applied in higher dimensional spaces without ever requiring the mapping to be explicitly calculated.

Since in general the problem of learning action effects is not linearly separable, the kernel perceptron is an appropriate choice for this problem. Kernel perceptrons obtain reasonable accuracy at acceptable training and prediction speeds, allowing this approach to be used in practical planning applications. Alternative non-linear classifiers, such as SVMs (Boser et al., 1992), can be substantially slower (Surdeanu and Ciaramita, 2007) while performance is not guaranteed to be better (Graepel et al., 2000). To improve the speed of the classifier I use a variant of the kernel perceptron, the *voted perceptron* (Freund and Schapire, 1999), which is computationally efficient and produces performance close to the best performing maximal-margin classifiers on similar problems. Furthermore, it is known to tolerate noise (Khardon and Wachman, 2007), essential for learning in noisy, partially observable domains.

The voted perceptron algorithm extends the perceptron algorithm so that instead of maintaining a single weight vector $\mathbf{w}$, it maintains a set of weight vectors $\{\mathbf{w}_k\}$. During training, a new weight vector $\mathbf{w}_k$ is created when a training example $\mathbf{x}_i$ is classified incorrectly. As well as setting $\mathbf{w}_k := \mathbf{w}_{k-1} + y\mathbf{x}_i$, the previous weight vector $\mathbf{w}_{k-1}$ is stored, along with c_{k-1}, the count of the number of correct predictions this weight vector made. At prediction, the decision function now takes the sign of the weighted average of the predictions made by each of the weight vectors:

$$f(\mathbf{x}) = sign(\sum_{i=1}^{n} c_i \, sign(\langle \mathbf{w}_i \cdot \mathbf{x} \rangle)).$$

As before, the kernel trick can be applied, giving a prediction function:

$$f(\mathbf{x}) = sign(\sum_{i=1}^{n} c_i \, sign(\sum_{j=1}^{i} \alpha_j y_j k(\mathbf{x}_j \cdot \mathbf{x})))$$

for some kernel function k. I adopt the terminology used by Freund and Schapire (1999), where the *support vectors* of the voted perceptron are those training examples which are used in the prediction calculation, that is, any $\mathbf{x}_i$ for which α_i is non-zero.

The voted perceptron has a parameter T which sets the number of passes which should be made through the training data. In all of the experiments presented in this thesis, $T = 1$ and so is omitted in the algorithm description in Figure 4.3.

```
Training:    Input:   Training examples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$
             Output:  Voted perceptrons $(\alpha_1, c_1), \ldots, (\alpha_n, c_n)$

  $k := 0, \alpha_i := 0, c_i := 0 \ \forall i$
  for $i = 1 \ldots n$ do
    $\hat{y}_i := sign(\sum_{j=1}^{i} \alpha_j y_j K(\mathbf{x}_j \cdot \mathbf{x}_i))$
    if $(\hat{y}_i = y_i)$ then
      $c_k = c_k + 1$
    else
      $\alpha_i := \alpha_i + 1$
      $c_i = 1$
      $k = i$

Prediction:  Input:   Unlabelled instance $\mathbf{x}$
                      Voted perceptrons $(\alpha_1, c_1), \ldots, (\alpha_n, c_n)$
             Output:  Prediction $\hat{y}$

  $\hat{y} := sign(\sum_{i=1}^{n} c_i \, sign(\sum_{j=1}^{i} \alpha_j y_j K(\mathbf{x}_j \cdot \mathbf{x})))$
```

Figure 4.3: Kernelised voted perceptron algorithm.

```
Training:    Input:   Training examples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$
             Output:  Parameter $\mathbf{M}$

               $\mathbf{M} := 0$
               for $i = 1 \ldots n$ do
                 $\hat{y}_i := \arg\max_{r=1}^{k} \langle \overline{M}_r, \mathbf{x}_i \rangle$
                 if $(\hat{y}_i \neq y_i)$ then
                   $\overline{M}_{\hat{y}} := \overline{M}_{\hat{y}} + \mathbf{x}_i$
                   $\overline{M}_{y_i} := \overline{M}_{y_i} - \mathbf{x}_i$

Prediction:  Input:   Unlabelled instance $\mathbf{x}$, Parameter $\mathbf{M}$
             Output:  Prediction $\hat{y}$

               $\hat{y} := \arg\max_{r=1}^{k} \langle \overline{M}_r, \mathbf{x} \rangle$
```

Figure 4.4: Multi-class perceptron algorithm in primal form

Some of the experimental domains include functions, so it is necessary to extend the voted perceptron to perform multi-class classification. A simple approach would be to decompose the multi-class classification problem into a set of binary classification problems. This approach leads to methods such as one-versus-all classification (each classifier separates one class from all the other classes) or all-pairs classification (each classifier separates two different classes). The disadvantages with these approaches are that many binary classifiers are needed to carry out a single multi-class classification, and some areas of the feature space can end up without a classification, or with multiple classifications.

Because of these disadvantages, the approach proposed by Singer and Crammer (2003) is taken instead. Here, the perceptron weight vector is replaced with a matrix **M**, with one row in the matrix for each class. At each step in the perceptron algorithm, the dot product is calculated for the current training instance x and each row in the matrix $\overline{M}_i$. The prediction for x is the j-th class where $j = \arg\max_i K(\bar{x}, \overline{M}_i)$. The update rule is *max-score multi-class update*, one of a family of update rules defined in Singer and Crammer (2003). The row of **M** corresponding to the correct class of $\bar{x}$ is updated with $\bar{x}$ while the row of **M** corresponding to the predicted class is updated with $-\bar{x}$. The multi-class perceptron algorithm is shown in Figure 4.4.

Voted perceptrons are linear classifiers, but can be kernelised to perform classification in higher dimensional feature spaces, or with structures such as graphs. Applying the kernel trick to the multi-class perceptron algorithm, note that for max-score multi-

```
Training:    Input:   Training examples $(\mathbf{x}_i, y_i)$ where $y_i \in \{1,\ldots,r\}$, $i \in \{1,\ldots,n\}$
             Output:  Voted perceptrons $(\{\alpha_{01},\ldots,\alpha_{0r}\}, c_0),\ldots,(\{\alpha_{n1},\ldots,\alpha_{nr}\}, c_n)$

  $\alpha_{ij} := 0, c_i := 0 \ \forall i, j$
  for $i = 1 \ldots n$ do
    $\hat{y}_i := \arg\max_{z \in GEN(\mathbf{x}_i)} \sum_{j=1}^{i} \alpha_{jz} K(\mathbf{x}_j, \mathbf{x}_i)$
    if $(\hat{y}_i = y_i)$ then
      $c_k = c_k + 1$
    else
      $\alpha_{i\hat{y}_i} := \alpha_{i\hat{y}_i} - 1$
      $\alpha_{iy_i} = \alpha_{iy_i} + 1$
      $k = i + 1$

Prediction:  Input:   Unlabelled instance $\mathbf{x}$
                      Voted perceptrons $(\{\alpha_{01},\ldots,\alpha_{0r}\}, c_0),\ldots,(\{\alpha_{n1},\ldots,\alpha_{nr}\}, c_n)$
             Output:  Prediction $\hat{y}$

  for $i = 1 \ldots n$ do
    $\hat{z} = \arg\max_{z \in GEN(\mathbf{x})} \sum_{j=1}^{i} \alpha_{jz} K(\mathbf{x}_j, \mathbf{x})$
    $votes_{\hat{z}} = votes_{\hat{z}} + c_i$
  $\hat{y} := \arg\max_{y \in GEN(\mathbf{x})} votes_y$
```

Figure 4.5: Kernelised multi-class voted perceptron algorithm.

class update, the row $\overline{M}_r$ is the sum of all the training examples where the r-th class was the target class, less the sum of all the training examples where the r-th class was the predicted class. Setting $\alpha_{ir} = 1$ if the r-th class was the target (but not predicted) class for example x_i, and $\alpha_{ir} = -1$ if the r-th class was the predicted (but not target) class for example x_i, then:

$$\langle \overline{M}_r, x \rangle = \langle \sum_{i=1}^{n} \alpha_{ir} x_i, x \rangle = \sum_{i=1}^{n} \alpha_{ir} \langle x_i, x \rangle.$$

The inner product $\langle x_i, x \rangle$ can now be replaced with a kernel function, namely, the kernel described below in Section 4.3.1. It is straightforward to extend the kernelised multi-class perceptron to include voting (Collins, 2002; Collins and Duffy, 2002). Following Collins (2002), we define a function GEN, which enumerates all observed values for a given element of the output vector. GEN thus provides the set of target classes. The resulting kernelised multi-class voted perceptron algorithm is shown in Figure 4.5.

4.3.1 Kernel function

A natural choice of kernel would be one which allows the perceptron algorithm to run over conjunctions of features in the original input space, as this permits a more accurate representation of the exact conjunction of features (action and preconditions) corresponding to a particular effect. A suitable kernel is the DNF kernel, $K(x,y) = 2^{same(x,y)}$, where $same(x,y)$ is the number of elements with the same value in both x and y (Sadohara, 2001; Khardon and Servedio, 2005). In the $same(x,y)$ calculation, any element with an unobserved value in x or y is considered to have a different value to the corresponding value in y or x. The DNF kernel has features which are all possible conjunctions of fluents. However, using the DNF kernel may scale poorly. It is an open question in computational learning theory whether DNF is PAC-learnable. What is known is that DNF is not PAC-learnable by a perceptron using the DNF kernel, as there exist examples on which it can make exponentially many mistakes (Khardon et al., 2005).

I therefore also consider the k-DNF kernel, whose features are all possible conjunctions of fluents of length $\leq k$ for some fixed k: $K(x,y) = \sum_{l=0}^{k} \binom{same(x,y)}{l}$ (Khardon and Servedio, 2005). An algorithm for PAC-learning k-DNF is known (Valiant, 1984). Furthermore, a perceptron using the k-DNF kernel has polynomial time updates (Sadohara, 2002) and a polynomial mistake bound (Klivans and Servedio, 2004). Since a mistake-bound algorithm can be converted into a PAC-learning algorithm (Angluin, 1988), k-DNF is also PAC-learnable by a perceptron using the k-DNF kernel.

4.4 Experiments

All experiments were run on the simulated data sets described in Chapter 3. Results were compared for a standard (non-kernelised) perceptron, a voted (non-kernelised) perceptron, and a voted kernel perceptron. Both the DNF kernel and k-DNF kernel with $k = 2,3$ and 5 were tested.

Where the domains are fully observable and noiseless, the classifiers are only learning simple conjunctions of features, and so it is expected that a standard perceptron is enough to learn the action models. The use of a voted perceptron is known to improve performance when there is classification noise (Khardon and Wachman, 2007), due to the majority vote mechanism, which stabilises the predictions. Since observations with attribute noise can be interpreted as noiseless observations with noisy classifications, it is expected that the voted perceptron will also perform better than the standard perceptron when there is attribute noise. Similarly, with partial observability, it is expected that a kernel whose features are conjunctions of fluents will improve performance. The redundancy in the representation means that small conjunctions of fluents can be combined to give high weight to a larger rule conjunction, even if all the fluents forming the full rule are never seen together in the training data.

4.4.1 Results

The fully observable, noiseless cases are easily learnt by any of the perceptrons tested, and after 5,000 training examples, the F-score[3] on the test set is 1, in almost all cases (results for the voted perceptron with the 3-DNF kernel are shown in Figure 4.7). Performance of the voted perceptron, with or without the various kernels, is almost identical (results not shown).

With the introduction of unobserved fluents or of noise, the voted perceptron performs better than the standard perceptron, as expected. However, the DNF kernel does not give improved performance, with the unkernelised voted perceptron learning significantly more accurate action models. In contrast, the k-DNF kernels all produce significantly more accurate models than with the DNF kernel or no kernel ($p < 0.05$)[4]. Figure 4.6 gives a comparison of the relative performance of each model.

Although there was no statistically significant difference between the 2-DNF and 3-DNF kernels at 20,000 actions, in some cases with lower numbers of actions the 2-DNF kernel produced worse results. This most likely reflects the 2-DNF kernel's lower redundancy and thus greater susceptibility to noise (see Section 4.5). Furthermore the k-DNF kernels are unable to learn the (k+1)-variable parity function,[5] and with lower values of k, the more likely it is that a rule may be a (k+1)-variable parity function. Therefore although the 2-DNF kernel is less complex, the 3-DNF kernel was selected for further experiments.

[3]F-score is the harmonic mean of precision and recall (true positives/predicted changes and true positives/actual changes, respectively) (Van Rijsbergen, 1979).

[4]Repeated measures ANOVA; post-hoc Bonferroni t-test

[5]For 1-DNF, i.e., the standard perceptron, this is the classic XOR problem.

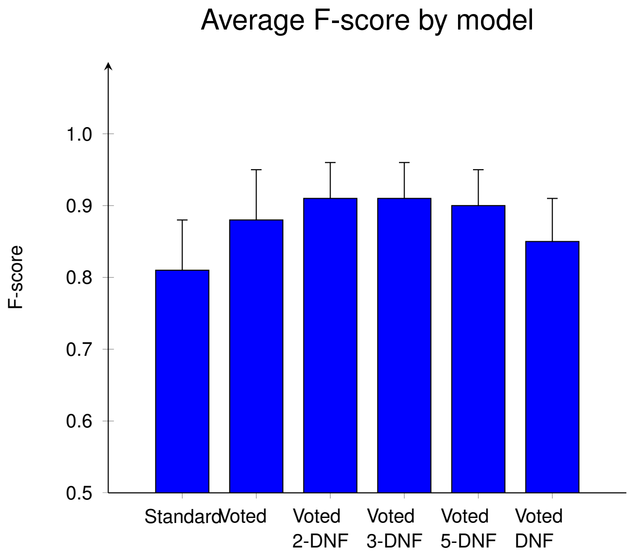


Figure 4.6: Comparison of the performance of different perceptron models on learning action models from 20,000 random actions in STRIPS domains, averaged across all domains, levels of noise and partial observability. Error bars are standard error. Performance is significantly different between models which use a k-DNF kernel and those which do not.

In the remainder of this section, the results of learning with the voted perceptron using the 3-DNF kernel are presented and discussed. Plots of the average F-scores for each domain, for different numbers of training examples, and at varying levels of noise and partial observability are shown in Figures 4.7, 4.8 and 4.9.

Overall, the accuracy of the models decreases with lower levels of observability and/or higher levels of noise. The rate of improvement of the F-scores decreases sharply after around 3,000 examples. This is likely to be a consequence of the random nature of the exploration of the domain, as the probability of seeing a useful example is much less than if the exploration were directed by some sort of active learning process.

Without noise, typically the learnt models will correctly predict when an action will be successful, and most of the fluents which change as a result, but not always when the action will fail, or all of the effects: that is, the models learn the correct effects but overly general preconditions. After 20000 examples, the F-scores for the pure STRIPS domains reach above 0.8, and in Rover, above 0.5, even when only 10% of the domain is observable. The F-scores for Rover are much lower than in the other domains, but this is unsurprising since Rover is a substantially larger domain, with conditional effects.

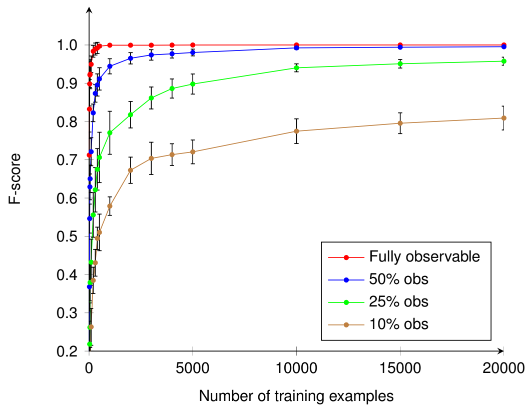
To give an idea of what this translates to in terms of the actual model, an example from the ZenoTravel domain, without noise and 10% observability, achieves an F-

Action	Over-general preconditions	Missing effects	Details
Board	2%	34%	over-general precondition for AT missing prediction for IN
Debark	-	34%	missing prediction for IN
Refuel	3%	-	over-general precondition for FUEL_LEVEL
Fly	6%	-	over-general precondition for AT
Zoom	7%	93%	over-general precondition for AT, FUEL_LEVEL missing prediction for AT

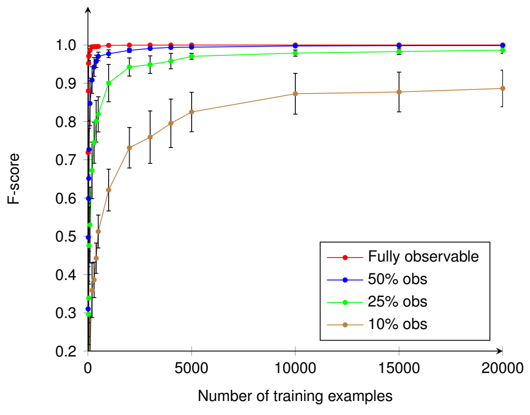
Table 4.1: Results for a typical ZenoTravel example with no noise and 10% observability. After 20000 training examples, the model achieves an F-score of 0.87. The errors in the preconditions and effects are shown above, indicating that the model will mostly correctly predict when any action is successful, but will miss some changes in its predictions for around a third of Board and Debark actions, and almost all Zoom actions.

score of 0.87, with errors broken down as shown in Table 4.1. In this case, around one-third of all Board and Debark actions will not have a change to the `(in arg1 arg2)` fluent predicted, while most of the Zoom actions will be missing a change to the `(at arg1 arg3)` fluent. In less than 10% of cases the Board, Refuel, Fly and Zoom actions will be wrongly predicted to change a small number of fluents, due to over-general preconditions. For both over-general preconditions and missing effects, the errors could (eventually) be corrected by further training.

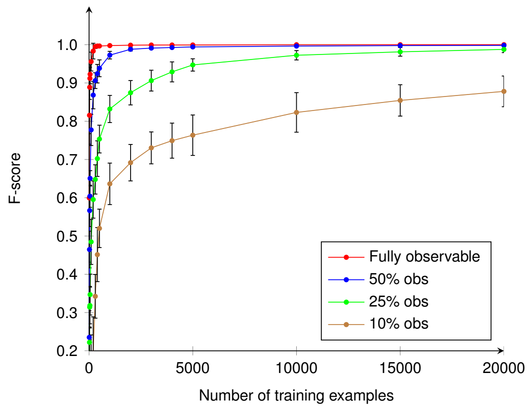
With the introduction of noise, predictions often also include erroneous effects, introduced by classification errors in the training data. This is most obvious in the results for the Rover domain, which has proportionally higher numbers of irrelevant fluents per object. After 20000 examples, the models learnt in noisy domains produce F-scores of 0.7 or more, even with 10% observability, except in the Rover domain. Here there appears to be a clear split in the results between the 25% and 10% observability levels, for both 1% and 5% noise. At 25% observability and above, some learning is taking place and the F-scores increase, albeit slowly. At 10% observability, the F-scores plateau: it seems the noise on irrelevant attributes drowns out the true action model.



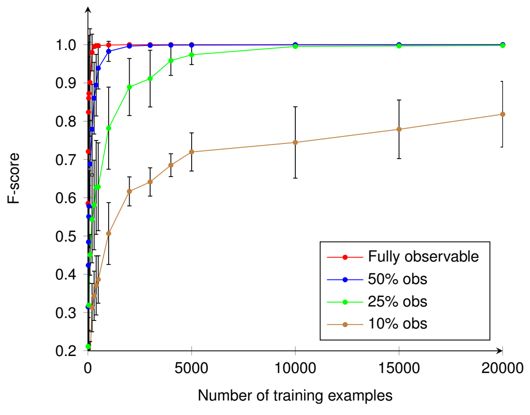
(a) BlocksWorld



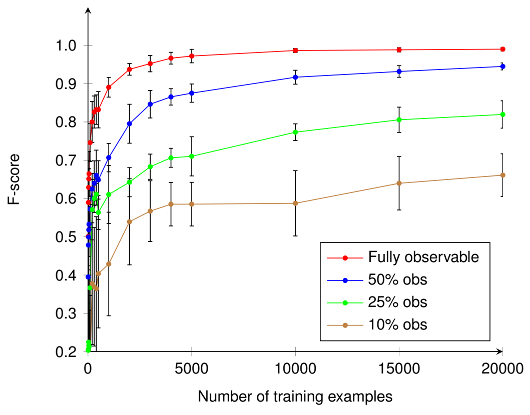
(b) ZenoTravel



(c) Depots

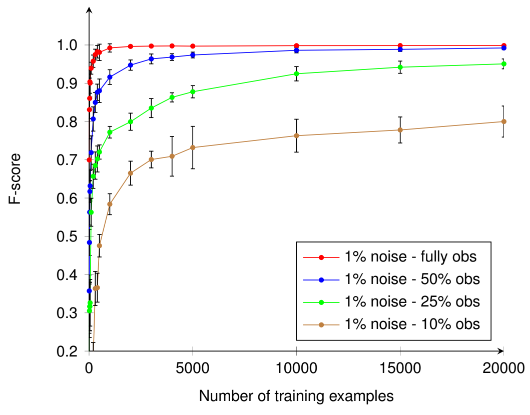


(d) DriverLog

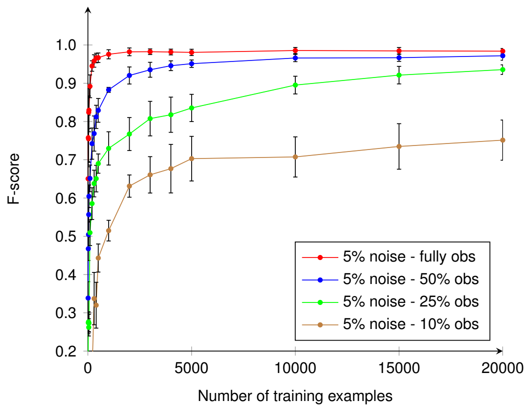


(e) Rover

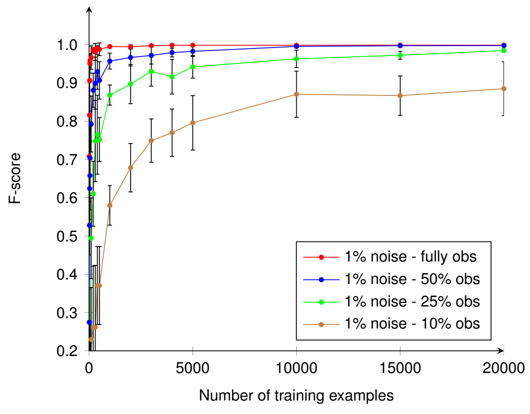
Figure 4.7: Results from learning actions in partially observable, noiseless, simulated planning domains, using a voted perceptron with the 3-DNF kernel. Classifiers were trained on varying numbers of examples, and tested on 2000 fully observed, noiseless examples from worlds in the same domain as the training examples, but with more objects.



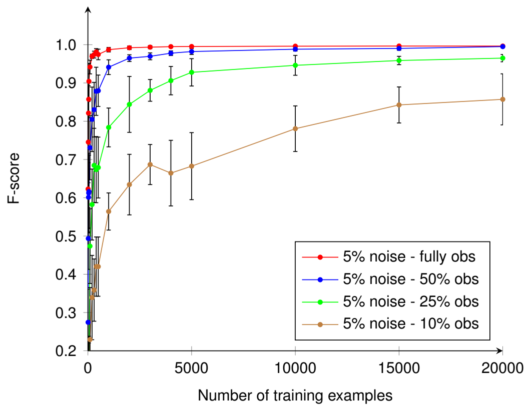
(a) Blocksworld (1% noise)



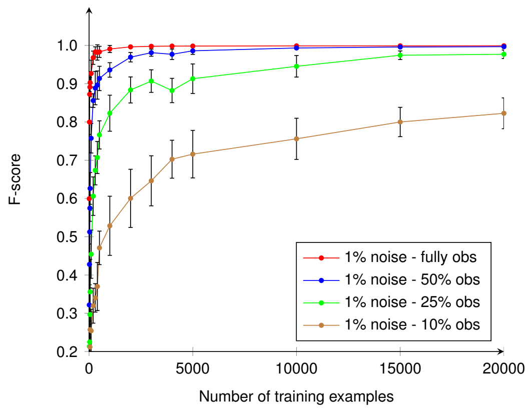
(b) Blocksworld (5% noise)



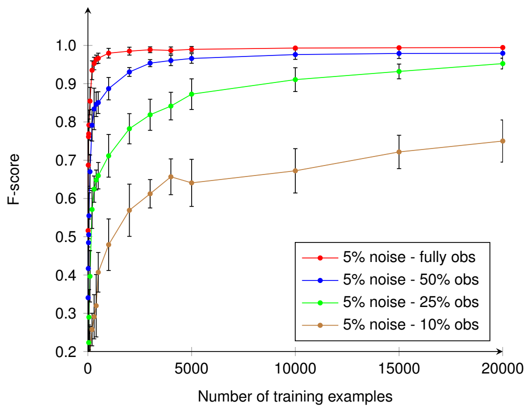
(c) ZenoTravel (1% noise)



(d) ZenoTravel (5% noise)

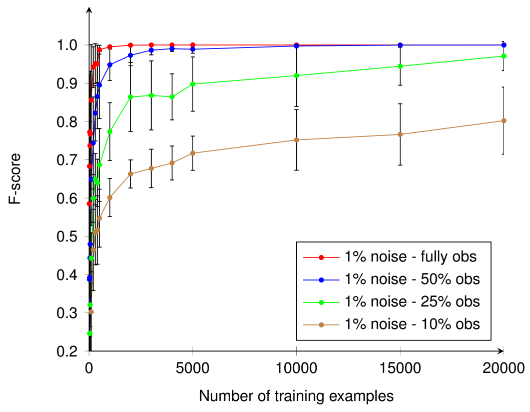


(e) Depots (1% noise)

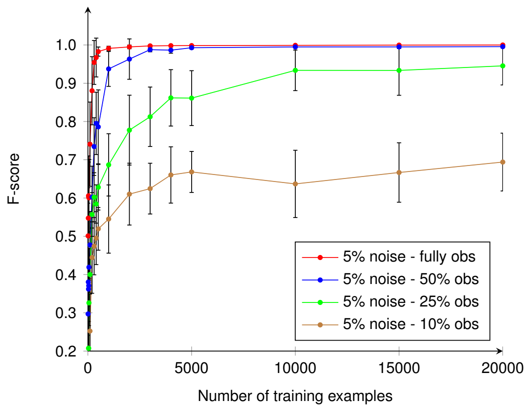


(f) Depots (5% noise)

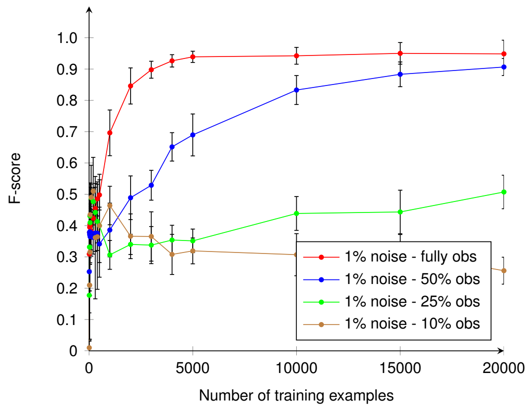
Figure 4.8: Results from learning actions in simulated planning domains with varying levels of noise (1%, 5%) and observability (100%, 50%, 25%, 10%), using a voted perceptron with the 3-DNF kernel.



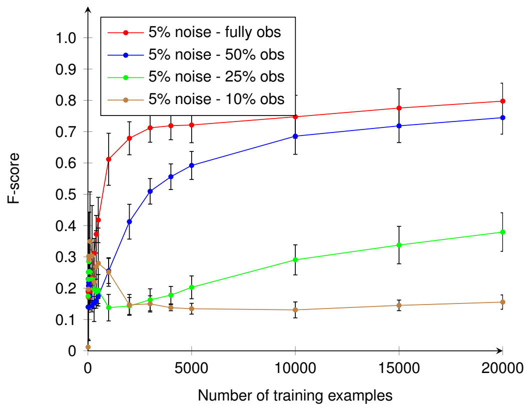
(a) DriverLog (1% noise)



(b) DriverLog (5% noise)



(c) Rover (1% noise)



(d) Rover (5% noise)

Figure 4.9: Results from learning actions in simulated planning domains with varying levels of noise (1%, 5%) and observability (100%, 50%, 25%, 10%).

4.5 Discussion

The experiments show that the action dynamics of fully observable, noiseless STRIPS domains can be learnt using standard perceptrons. Such a result is to be expected since there is a known algorithm for learning pure STRIPS action models in noiseless, fully observable domains (Walsh and Littman, 2008). The experiments also show that by using a voted, kernelised perceptron, the speed and accuracy of learning in these domains can be improved, and that action models can still be learnt in partially observable, noisy domains.

The results for the different kernels demonstrate a trade-off not only between expressivity and computational efficiency, but also robustness to noise and partial observability versus computational efficiency. Increasing the maximum lengths of feature conjunctions allowed by a kernel clearly increases its expressivity, since the space of possible hypotheses is increased, at the expense of greater computational effort. However, by allowing features to effectively overlap each other, the kernels also introduce a degree of redundancy, which improves learning in noisy and partially observable cases, and leads to the improved accuracy of the k-DNF kernels relative to the unkernelised perceptrons. Conversely, the costs of the larger search space of the full DNF kernel outweigh the benefit of the redundant features, and so its accuracy is lower than the k-DNF kernels.

4.5.1 Relation to other approaches in action dynamics learning

The structure of the learning model presented above is similar to the models used by Halbritter and Geibel (2007) and Croonenborghs et al. (2007). They also decompose the problem and use classifiers that learn to predict subsets of the action effects. However, both methods predict changes to individual *predicate symbols*, rather than fluents, using SVM classifiers and relational probability trees respectively.

Learning via a classifier assigned to each predicate is more difficult than learning via a classifier assigned to each instance of a fluent. Not only must a predicate classifier learn to predict for each instance of a fluent involving that predicate, it must also learn to differentiate between fluents with different predicted changes. Conversely, a fluent classifier only learns to predict for its particular instance, with the distinction between different instances handled by the structure of the model. For instance, in BlocksWorld, a predicate classifier learns when `(on x y)` changes for any x and y in the set of action parameters $\{arg1, arg2\}$. The input to the classifier is the action and the state description, which must be augmented with an additional input indicating which of $arg1$ and $arg2$ is x, and which is y. In this case there are two corresponding fluent classifiers, respectively predicting when `(on arg1 arg2)` and `(on arg2 arg1)` change. The choice between learning to distinguish between different instances of fluents, and building the distinction into the model structure, becomes more important when working with representations beyond STRIPS, and is discussed in Chapter 5.

Furthermore, Halbritter and Geibel (2007) use a separate classifier to predict whether any action's preconditions are met, and only use the remaining classifiers to predict the effect of the action, given that it is known to be successful. They choose this structure so that when predicting for an action whose preconditions are unsatisfied, only one SVM calculation is needed, rather than a series of potentially expensive prediction calculations for each type of relation in the world. Croonenborghs et al. (2007) have an additional binary random variable which performs a similar function. Note that the single "successful action" classifier is in effect doing most of the work, as in the absence of conditional effects the remainder of any model is just mapping from action type to change. The disadvantage of the alternative structure used by Halbritter and Geibel, and Croonenborghs et al. is that conditional effects cannot be modelled, since the same precondition is assumed to apply to every effect of an action.

The classification method used by Halbritter and Geibel is clearly also similar, since the voted perceptron could easily be replaced by an SVM using the same kernel. However, their representation is graph-based and so they use a graph kernel (the product graph kernel). While this kernel is suitable for STRIPS domains, it is not expressive enough to capture the rules underlying more complex domains. The preconditions are limited to those which can be represented as a walk in the graph representation of the state, namely, a conjunction of positive fluents without universal quantification. (We return to this point in Chapter 5, section 5.3.3.2.) The effects which can be learnt are restricted by their assumption that nodes affected by an action must be listed in the action parameter list. Without this artificial restriction, effects with universal quantifiers could be learnt, although with extra computational cost. Consequently, despite the additional machinery, Halbritter and Geibel's method does not learn more than STRIPS action models, without conditional effects or negative preconditions.

Most existing methods for learning action models in partially observable domains (Shahaf and Amir, 2006; Yang et al., 2007; Amir and Chang, 2008; Zhuo et al., 2010) operate in different settings to the one used above, and do not handle noise. The ARMS algorithm (Yang et al., 2007) learns STRIPS action models, but it learns from partially observed plan traces. LAMP (Zhuo et al., 2010) learns action models in domains more complex than STRIPS, again from partially observed plan traces. Both algorithms depend on the assumption that when an action appears in a plan, its preconditions are met, so noisy observations and action failures are not permitted. ARMS and LAMP can therefore not be applied to the learning problem above. Conversely, my method requires action failures in order to learn preconditions, and so is not applicable in the ARMS/LAMP setting. Similarly, SLAF (Amir and Chang, 2008; Shahaf and Amir, 2006) tractably learns STRIPS action models in partially observable domains, but assumes that either actions do not fail, or that the algorithm is provided with an indicator that the action preconditions were not met. With these restrictions, SLAF only learns action effects, and preconditions are generated via a heuristic. Also, noisy observations are not permitted. Again, therefore, SLAF cannot be applied to the learning problem above.

The exception is the LE_a algorithm (Bouthinon et al., 2009), an attribute-value learner which learns concepts (in DNF) from partially observed examples. Since the vector representation effectively reduces the states to an attribute-value representation, LE_a could be applied to learning action models in the framework above. However, the method is intolerant to noise.

Rodrigues et al. (2010a) take an Inductive Logic Programming (ILP) approach to learning action models from noisy examples. They can represent STRIPS rules, and also more complex forms, such as where preconditions are disjunctions of conjunctions, or where objects listed in the preconditions or effects are not in the action parameters. Negative preconditions are not supported. Since they use the learning from interpretations setting (De Raedt and Džeroski, 1994), the method does not directly support partial observability, as learning from interpretations assumes that there are no missing values. Their definition of noise also differs significantly: the percentage noise parameter refers to the probability that an observed state is subject to *any* noise, and an additional parameter n_ε specifies how many predicates in the state are affected. In their experiments, n_ε is set to 2. Their training data is therefore substantially less noisy than in the experiments presented earlier in this chapter. The main benefit of the ILP approach over the classification approach is that rules are directly available after learning. However, to limit the effects of noisy examples on the learnt rules, the algorithm requires several parameters whose best values are likely to vary with the domain and level of noise. Furthermore, the approach is likely to scale poorly, because it learns full DNF rules, and there are no steps to reduce the number of objects, or limit the number of literals in each conjunct of the DNF hypotheses.

4.5.2 Limitations

The STRIPS scope assumption (Section 2.2.2) is very strong, since it is unlikely that an agent will have actions conveniently defined with the relevant objects in the action signature. Relaxing the assumption, by extending deictic references beyond the action parameters, means that the deictic terms have a hierarchical structure, ordered by a generality relation. For example, in the ZenoTravel domain, if $arg1$ and $arg2$ are action parameters, the deictic terms $\{x : \texttt{(in arg1 x)} \wedge \texttt{(at x arg2)}\}$ and

$\{x : \neg$ `(in arg1 x)` $\wedge$ `(at x arg2)`$\}$ have a common, more general, deictic term $\{x :$ `(at x arg2)`$\}$. Whereas in the STRIPS case, objects with the same role in an action had identical constraints in the deictic terms, now they have intersecting constraints, which poses a problem for the kernel calculation, as decribed below.

The kernel calculation on the vector representation relies on the assumption that entries in the same position in different vectors are comparable, so that comparing individual entries can form the basis of a similarity comparison. This in turn depends on objects with the same role in an action occupying the same relative positions in the state vectors. This positioning is supported under STRIPS, because there is a 1-1 mapping between deictic terms in different states. However, this is no longer possible when a common role for two objects is only indicated by intersecting rather than equal contraints in their deictic terms: the deictic term constraint of an object in one state may intersect with several different objects' constraints in another state. For instance, in the ZenoTravel example above, an object with deictic term $\{x :$`(in arg1 x)` $\wedge$ `(at x arg2)` $\}$ matches objects in another state with terms $\{x :$`(in arg1 x)` $\wedge\neg$`(at x arg2)` $\}$, or $\{x : \neg$`(in arg1 x)` $\wedge$ `(at x arg2)` $\}$.

In some simple cases, the problem can be handled by explicitly defining slots of the vector for each possible deictic term. The model only manages to learn the action dynamics by learning a separate rule for each possible deictic term, and is unable to generalise across the deictic terms. Although this generalisation could be attempted as a post-process, in larger domains, it is impractical to learn separate rules for all possible combinations of some subset of deictic terms, as there will be exponentially many potential rules to learn. Furthermore, having a slot in the vector for each deictic term breaks down under partial observability, as the position of a partially observed object in the vector becomes undefined. Ultimately, a different representation is needed to capture the relational structure of the state descriptions when they include more general deictic terms. This is the subject of Chapter 5.

An additional limitation is that the models learnt in this chapter do not produce action definitions as explicit planning rules, but rather behave as a black box which must be queried to discover what effects, if any, an action is predicted to have. This restriction makes it difficult to integrate the models with current planners, which expect rules as input. The learnt models could be used as the world model in model-based reinforcement learning, e.g. in Dyna (Sutton, 1990). Alternatively rules can be extracted from the models, as discussed in Chapter 6.

Finally, noise on irrelevant attributes activates classifiers on fluents which in noiseless domains are never used. This affects the time to train the models (more classifiers must be updated), and the accuracy of the results (many predicted changes are incorrect). When there are many fluents describing the domain, such as in the Rovers domain, and low levels of observability, there will be relatively fewer training examples for the affected classifiers and so they are more likely to continue to make incorrect predictions. An improvement to the accuracy would be to ignore the predictions of classifiers with low reliability, or to track reliability against predictions of no change, and choose the best one. A partial implementation of this solution is discussed in Section 5.1.3. A more accurate noise model (Section 7.2.1.2) might also improve these results since noise is likely to be concentrated on particular fluents rather than uniformly distributed across the fluents. Depending on the overlap between the actual action model and the noisy fluents, fewer incorrectly predicted fluents should lead to higher F-scores.

4.6 Summary

The problem of learning STRIPS action models can be converted into a set of classification problems which can be learnt by standard classifiers, such as the perceptron. The approach depends on the (strong) STRIPS scope assumption that it is known which objects are relevant to an action, and what their relative roles in the action are. By incorporating voting and using the k-DNF kernel, for some small k, noise and partial observability in the world state can be handled by the perceptron learning model.

The same approach can also be used for extended deterministic STRIPS models, where preconditions may be in DNF, although these are much harder to learn. For tractability it is necessary to assume that the number of fluents in a precondition is bounded by some constant.

Changes are needed if the approach is to be applied in domains where the STRIPS scope assumption does not apply. The learning model will not be able to use an attribute-value representation for world state. Crucially, this means that it will not be possible to uniquely map objects which have the same roles in different states. It follows that the kernels used in this chapter cannot be applied. Therefore both a new representation and a new similarity measure are needed. Learning in such domains is the focus of the next chapter.

Chapter 5

Learning action models beyond STRIPS

In the previous chapter, the problem of effects learning was structured as a set of classification problems on vectors representing reduced descriptions of the world state. The model easily learns the effects of actions in classical STRIPS planning domains, even when partially observable or noisy. However, the vector representation fails in partially observable or noisy domains where the STRIPS scope assumption does not apply (Section 4.5.2). Comparisons between states can no longer rely on the vector to encode a map between objects with the same role in a state. In this chapter, I introduce an alternative graphical representation of the world state. I show how this representation fits into the existing model framework and present results using the new representation with noisy and partially observable STRIPS and non-STRIPS domains.

5.1 Moving to a graphical representation

With the earlier vector representation, learning proceeds as follows:

- reduce the descriptions of the world state before and after an action to the set of predicates involving only objects which are action parameters;
- represent the resulting reduced states as vectors, and generate an effect vector by calculating the difference between the state vectors before and after the action;
- learn each element of the effect vector as a separate classification problem using voted kernel perceptrons with a k-DNF kernel;
- make predictions with the model or extract rules.

The main problem with the vector representation is that it imposes ordering constraints on the objects represented in the vector. These can be avoided by using a graphical representation instead. The learning mechanism must now itself determine which objects have similar roles in each action, rather than relying on the positioning within the state vector. Moving to a graphical representation entails not only changing the representation, but also replacing the k-DNF kernel with a graph kernel function, and determining the *unit of prediction* — the quantity that each classifier will predict.

5.1.1 Graphical representations of world state

Graphs are a natural way to represent objects and relations between them. Van Otterlo (2009) discusses a range of applications of graphical representations to relational learning. Of these, only *relational attribute graphs* are applied to representations of world state (Gärtner et al., 2003a; Driessens et al., 2006; Dabney and McGovern, 2006; Halbritter and Geibel, 2007). In relational attribute graphs, the world state is represented as a labelled directed graph with objects as nodes and instances of relations between objects represented as edges. Edges are labelled with the name of the corresponding relation, and nodes with properties of the corresponding objects. Gärtner et al. (2003a) and Driessens et al. (2006) go further and encode the next action and its outcome in the graph, by treating objects before and after the action as different objects, and inserting special edges between them, labelled with the action name. Edges may additionally be labelled with the argument position, in order to disambiguate the edges when the corresponding relation has more than two arguments (Gardiol and Kaelbling, 2007).

The drawback with relational attribute graphs is that the representation can be ambiguous. For example, the graph shown in Figure 5.1 depicts two instances of the ternary relation r, with the edges linking the first and second arguments labelled r_1, and the edges linking the second and third arguments labelled r_2. However, this could represent either the pair of relations $r(o1,o3,o4)$ and $r(o2,o3,o5)$, or the pair $r(o1,o3,o5)$ and $r(o2,o3,o4)$. The problem is that the representation does not uniquely distinguish between all instances of the relation $r(*,o3,*)$ (where $*$ is a wildcard which can be filled by any object in the world).

To avoid the ambiguity, the representation must distinguish between relations (a set of tuples in a Cartesian product space) and instances of that relation (a specific tuple). This may be achieved by generalising relational attribute graphs to oriented

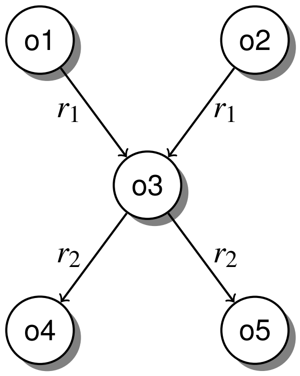


Figure 5.1: An example of an ambiguous relational attribute graph. Edges are labelled with the relation name (r) and the argument position. The graph could represent either the pair of relations $r(o1, o3, o4)$ and $r(o2, o3, o5)$ or the pair $r(o1, o3, o5)$ and $r(o2, o3, o4)$.

hypergraphs. A hypergraph is a collection of a finite number of vertices V and edges E, where edges can link more than two vertices; in an oriented hypergraph the vertices linked by an edge are ordered, so $E \subseteq \bigcup_{k=1}^{|V|} V^k$. A hypergraph representing a world state has a vertex for each object in the world, and an edge for each fluent. Having a unique edge for each fluent ensures that each instance of a relation is unambiguously represented in the hypergraph. It is trivial to extend the notation to include actions, by treating an action on the state in exactly the same way as a fluent, with an edge for the action containing an ordered tuple of the action parameters.

Oriented hypergraphs can be converted to an equivalent bipartite graph $G = \langle V \cup E, E' \rangle$ where $E' = \{(v, e) : v \in e\}$ and the edge (v, e) is labelled with i iff v is the i-th element of e. In the bipartite graph, objects are represented by nodes, as before, but now instances of relations are also represented by nodes. For example, in the BlocksWorld domain, with two objects A and B, the action `(unstack A B)`, in a state where block A is on block B, block A is clear, and block B on the table, would be represented as in Figure 5.2. In the remainder of this chapter, bipartite graphs are used to represent world states, or world states combined with an action.[1] These state graphs are formally defined below.

[1]This graphical representation is close to the working memory representation used in the Soar cognitive architecture (Laird, 2008).

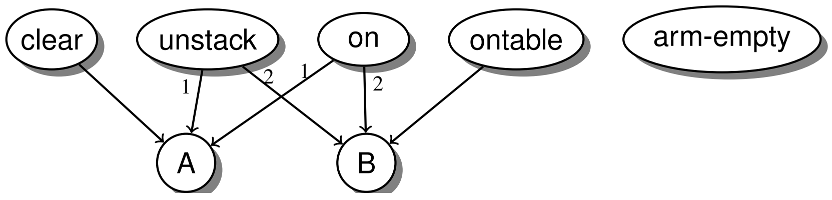


Figure 5.2: Graphical representation of a state in the BlocksWorld domain. Here, block B is on the table, block A is on block B, block A is clear, the gripper is empty, and the action to be performed is `(unstack A B)`.

Definition 5.1.1. Recall that a *domain* $\mathcal{D}$ is defined as a tuple $\mathcal{D} = \langle O, \mathcal{P}, \mathcal{F}, \mathcal{A} \rangle$, where 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. For a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, the *state graph* is the bipartite graph $G = \langle R \cup C, E \rangle$ where

- $R = s \cup \{a\}$ is the set of *fluent nodes*,
- $C = \{c : \exists \phi \in R, c \in args(\phi)\}$ is the set of *object nodes*, and
- $E = \{(\phi, c) : \phi \in R \wedge c \in args(\phi)\}$ are edges indicating which relations apply to which objects.

Nodes and edges of the state graph are labelled by a labelling function *label*. Fluent nodes are labelled with the corresponding predicate or action symbols, and object nodes are labelled with the object name. Where an edge (ϕ, c) links a fluent node to an object node, the edge label is an integer i, if c is the i-th argument of ϕ.

This definition of state graph follows from a closed world assumption, and so negative fluents do not need to be represented in R. As with the vector representation under partial observability, using the closed world assumption with this representation would make it impossible to distinguish between unobserved fluents and negative fluents, adding unnecessary noise to target classes and making it difficult to learn negative preconditions. Discarding the closed world assumption, I redefine a state s as any set of negated or unnegated fluent expressions, subject to the constraint that if $x \in s$ then $\neg x \notin s$. The types of the arguments of negated fluents must match the type signature of the fluent, to prevent negations of fluents which could never occur. Any (legal) fluent expression not in s is unobserved. Unfortunately, states meeting this definition could have a number of fluents exponential in the number of objects in the world, and so the state size is reduced by filtering out objects via deictic reference, discussed below.

5.1.1.1 The role of deictic reference

As before, deictic references are used to support generalisation across states, as well as to reduce the set of objects considered by the learning model. In Chapter 4, deictic references were restricted so that an object only had a deictic term if it was an action parameter. This restriction was possible because of the STRIPS scope assumption. Here I make the *deictic scope assumption* that objects mentioned in the preconditions or effects are either action parameters[2] or related to the action parameters. In this work we restrict ourselves to a *first-order* deictic scope assumption, where related objects must be directly related to the action parameters.[3]

Additionally, we add the constraint that for an object to have a deictic reference, it must be linked by a positive fluent to either an action parameter, or another object which has a deictic reference (the *positive link assumption*). This additional restriction accounts for the open world representation now in place, 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.

Every action parameter has its own unique deictic term, corresponding to its position in the parameter list, while the deictic terms of other objects are their definitions in terms of their relations with the action parameters. For example, in the Briefcase domain (Figure 5.3a) if the action were `(move L1 L2)` in the state shown in Figure 5.3b, $L1$ and $L2$, as action parameters, would have deictic terms $arg1$ and $arg2$ indicating their positions in the *move* action argument list. Relative to the `(move L1 L2)` action, object A is referred to by the deictic terms $x : at(x, arg1)$, $x : in(x)$, $x : \neg at(x, arg2)$, and any conjunction of these. Object H has no positive relations with the action parameters or other objects with deictic references, and so has no deictic reference in this case.

Apart from the action parameters, any object in a state may be referred to by several deictic terms, and any deictic term may refer to several objects in a state. However, since only deictic terms will be used to describe states for the learning algorithm, objects with an identical set of deictic terms will have identical predictions. Thus it is only necessary to consider one instance of each deictic term, by working with equivalence classes of objects under deictic reference, defined as follows.

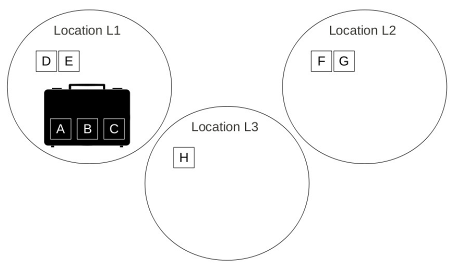
Recall (Definition 3.2.3) that deictic terms partition the set of objects in a state into a set of equivalence classes, where any two members of an equivalence class share

[2]Thus the STRIPS scope assumption is a special case.

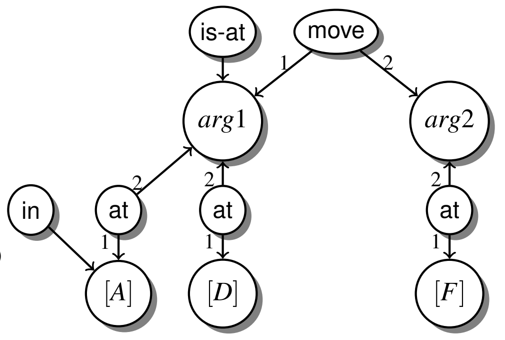
[3]Higher-order deictic references are possible, where objects are 2, 3 or more steps from the action parameters, but this is left to future work.

```
(define (domain briefcase)
  (:requirements :adl)
  (:types portable location)
  (:predicates (at ?y - portable ?x - location)
               (in ?x - portable)
               (is-at ?x - location))

(:action move
  :parameters (?m ?l - location)
  :precondition  (is-at ?m)
  :effect (and (is-at ?l) (not (is-at ?m))
      (forall (?x - portable) (when (in ?x)
        (and (at ?x ?l) (not (at ?x ?m)))))))

(:action take-out
  :parameters (?x - portable)
  :precondition (in ?x)
  :effect (not (in ?x)))

(:action put-in
  :parameters (?x - portable ?l - location)
  :precondition (and (not (in ?x)) (at ?x ?l) (is-at ?l))
  :effect (in ?x)))
```

(a)



(b)



(c)

Figure 5.3: (a) A PDDL description of the Briefcase domain, (b) a state in the Briefcase domain, and (c) its graphical representation (as a situation graph) when combined with the `move` action. Objects are represented by their deictic terms: here, given the action `(move arg1 arg2)`, `[A]=`$\{$`x:(at x arg1)` $\wedge$ `(in x)` $\wedge$ $\neg$`(at x arg2)`$\}$, `[D]=`$\{$`x:(at x arg1)` $\wedge$ $\neg$`(in x)` $\wedge$ $\neg$`(at x arg2)`$\}$, and `[F]=`$\{$`x:(at x arg2)` $\wedge$ $\neg$`(in x)` $\wedge$ $\neg$`(at x arg1)`$\}$. For clarity, negative relations are omitted in the graph.

the same set of deictic terms. Then $x_1 \sim x_2$ if every deictic term which refers to object x_1 also refers to x_2 and vice versa. Similarly, deictic terms also partition the set of fluents in a state into a set of equivalence classes where for $\phi_1, \phi_2 \in s$, $\phi_1 \sim \phi_2$ iff $label(\phi_1) = label(\phi_2)$ and $\forall i\, args_i(\phi_1) \sim args_i(\phi_2)$. Extending the notion of arguments to the fluent equivalence classes, $args_i([\phi_1]) = [args_i(\phi_1)]$ and $args([\phi_1]) = \bigcup_i \{[args_i(\phi_1)]\}$.

By representing objects by their equivalence classes, a reduced form of the state graph can be constructed, as in Figure 5.3b. It is an underlying assumption of the learning procedure that this core *situation* encompasses the relevant information for learning the action model.Thus states of the world are represented by *situation graphs*, where nodes in the graph represent either the current action, or the equivalence classes of fluents and objects defined above. Negated fluents are also included. Fluent nodes are labelled with the corresponding predicate or action symbols, and object nodes with the object name of a representative in the equivalence class. Edges link equivalence classes of fluents (or the current action) and their arguments, and are labelled with the argument position.

Definition 5.1.2. For a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, the *situation graph* is the bipartite graph $G = \langle R \cup O, E \rangle$ where

- the set of fluent nodes is R, where
  $R = \{[r] : [r] = \{x : x \in s \wedge x \sim r \wedge args(x) \cap args(a) \neq \varnothing\}\} \cup \{a\}$,
- the set of object nodes is O, where
  $O = \{[c] : \exists [r] \in R$ such that $[c] \in args([r])\}$, and
- the set of edges is $E = \{([r],[c]) : [r] \in R \wedge [c] \in O \wedge [c] \in args([r])\}$.

Figure 5.3c shows a situation graph for the state depicted in Figure 5.3b in the context of the `(move L1 L2)` action. The object equivalence classes are $[arg1], [arg2], [A], [D]$ and $[F]$, since $A \sim B \sim C$, $D \sim E$ and $F \sim G$.

5.1.2 Structure of the learning model

Using the situation graphs defined above, the structure of the learning model can now be defined. Given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, the model is to predict the successor state s'. Equivalently, the set of fluents which change between s and s' — the deltas — can be predicted. Since it is assumed that each situation contains enough information to learn the model, it is sufficient to predict deltas for situation graphs rather than full state graphs. The strategy employed here is to first construct the situation graph from the state, and then decompose the set of all possible fluent nodes into subsets (*units of prediction*), each of which is associated with a separate classifier. Each classifier predicts the delta for its subset of relations, given an input situation and an action. The final prediction of the full delta is generated by combining (by conjunction) the predictions of all the classifiers.

5.1.2.1 Unit of prediction

It makes sense to split the prediction problem along the dimensions of action, and type of relation, since predictions are likely to be different for different actions and types of relations. Without further decomposition, this amounts to a classifier per type of relation. To make this work, an additional input is needed, to specify which particular instance of a relation is being predicted, that is, to specify which objects are parameters

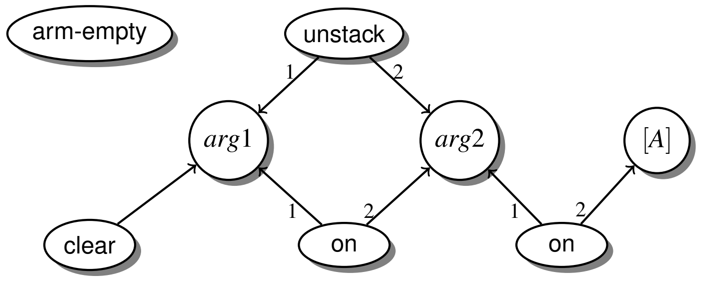


Figure 5.4: BlocksWorld `unstack` example (only positive fluents shown).

of the relation. For example, if the situation graph shown in Figure 5.4 was input to an `on` classifier, it would be necessary to indicate whether the classifier is to predict change to `(on arg1 arg2)` or to `(on arg2 [A])`.

Alternatively, the problem can be decomposed further, with a classifier for every possible instance of a relation. Then for the situation graph in Figure 5.4 there would be a separate classifier for each of the `on` nodes and no further differentiation would be needed. However, there remains a similar problem: if there are several classifiers which predicted different `on` relations in previous examples, which one is to predict the change to `(on arg1 arg2)` and which to `(on arg2 [A])`? In either case, the problem is of consistently applying the classifiers to different input situations, namely that of identifying a mapping between instances of relations in different situations, so that the classifier is always assigned to objects playing the same role.

Using deictic reference resolves the problem by providing a mapping between objects in different situations: objects can be mapped to each other if their deictic terms share a constraint. Note that, to be mapped, the deictic terms need not be identical; only their constraints need intersect. This leads to two possible algorithm structures, based on the discussion above. Firstly, for each action and type of relation, there is a classifier which as well as taking a state and an action as input, also takes a combination of deictic terms which specify the particular relation instance(s) it is to predict. In the `unstack` example above, there would be a single classifier for the `unstack-on` case, and the deictic term input could take the form of `(arg1,arg2)`, `(arg2,arg1)`, or `(arg1,{x:(on arg2 x)})`, etc., corresponding to requests to predict change to `(on arg1 arg2)`, `(on arg2 arg1)` or `(on arg1 {x:(on arg2 x)})` respectively. Secondly, for each action and type of relation, there is a classifier for each possible combination of deictic terms which can form the arguments of the relation. This structure corresponds to a classifier for each possible node in any situation graph. In the `unstack` example above, there would be classifiers for each of the relation instances `(on arg1 arg2)`, `(on arg2 arg1)`, `(on arg1 {x:(on arg2 x)})`, etc.

The first option has relatively fewer classifiers but the classification problem each has is more complex. Each classifier must not only learn when a state affords a particular action, but also to differentiate between different instances of a relation. Conversely, the second option has many more classifiers, but each classifier only has to learn whether its one instance of a relation changes when a state affords an action. The first option involves learning part of the problem structure which is already known. In the process it is likely to make many mistakes, which affects the time to learn the prediction problem. The second option is therefore likely to be faster even although it requires training more classifiers. Furthermore, the overhead in training is the kernel calculation, whose results can be cached and re-used. Therefore in the following sections the learning model is structured according to the second option, with one classifier per instance of a relation.

5.1.2.2 Learning algorithm

Actually generating a classifier for every possible node in a situation graph is undesirable, so classifiers are only instantiated when the training data requires it. Figure 5.5 describes the process, as follows. The algorithm is provided with a set of training examples, each consisting of a state description x_i, an action a_i, and a successor state $x_i{}'$. Both state descriptions are converted into situation graphs, based on the action a_i, as described earlier. The algorithm also initially knows a set of action labels A, a set of predicates P, and the number and types of their arguments.

The deltas from the training examples provide target values for each classifier. In a fully observable, noiseless domain, it is trivial to calculate the delta y_i for an example, by comparing the two *state* graphs and identifying which fluents changed. This depends on the underlying assumption that objects with the same deictic term will be affected in the same way by an action. However, objects which are observed to have the same deictic term may not be (or appear to be) affected in the same way by an action operating in a partially observable or noisy domain. To handle this, the assumption is weakened, and it is assumed only that most objects with the same deictic term will be affected in the same way by an action. The delta for a specific deictic term is then calculated by tracking all the objects with that deictic term in the prior state, and finding the corresponding deictic term(s) in the successor state. The “true” successor deictic term is assumed to be the one to which the most objects are mapped.

```
Training:  Input:   Training examples $X = \{(x_1, a_1, \delta_1), \ldots, (x_n, a_n, \delta_n)\}$,
                    Known action labels $A$, known predicates $P$
           Output:  Model parameters

Training:
  $C_{a,p} := \varnothing \;\; \forall a \in A, \forall p \in P$
  for all $(x, a, \delta) \in X$ do
    for all $p \in P$ do
      $M := findArguments(x, p)$
      for all $m \in M$ do
        if $p(m) \in \delta$ then
          $y^m := 1$
        else
          $y^m := 0$
        call $findClassifier(C_{a,p}, x^m, y^m)$

procedure $findClassifier(C, x^m, y^m)$
  $exactMatch := false$
  $intersectMatches := \varnothing$
  for all $c \in C$ do
    if $deicticRefs(x^m) \supseteq deicticRefs(c)$ then
      call $learn(c, (x^m, y^m))$
      call $updateReliability(c)$

    if $deicticRefs(x^m) = deicticRefs(c)$ then
      $exactMatch := true$
    else if $deicticRefs(x^m) \cap deicticRefs(c) \neq \varnothing$ then
      $intersectMatches := intersectMatches \cup \{c\}$

  if $(y^m \neq 0) \wedge (exactMatch = false)$ then
    $c_{new} := createNewClassifier(x^m)$
    $C := C \cup \{c_{new}\}$
    for all $i \in intersectMatches$ do
      $c_{new} := createNewClassifier(x^m, i)$
      $C := C \cup \{c_{new}\}$
```

Figure 5.5: Outline learning algorithm: each training example is a situation graph x with the action a which was performed, and a target value δ giving the changes which were observed.

The algorithm builds a set of classifiers $C_{a,p}$ for each action a and each predicate p. Every member of $C_{a,p}$ will be a classifier for a different combination of deictic terms (which are valid arguments of p). The classifiers are constructed as required during training.

Given a training example, each predicate p in the domain is considered in turn. A set M of all possible combinations of objects in example x_i which could be arguments of p is generated ($findArguments$). Objects are represented in M by their deictic terms. Often the potential set of arguments can be limited by considering the types of the deictic terms, and the types of the predicate's arguments. For each combination of deictic terms $m \in M$, the current set of classifiers $C_{a,p}$ is searched for any classifier which predicts for $p(m)$ (*findClassifier*). There may be more than one matching classifier, as some deictic terms can be more general than others. For example, consider the Briefcase domain deictic terms `dr1=`$\{$`x:(at x arg1)`$\}$ and `dr2=`$\{$`x:(at x arg1)` $\wedge \neg$`(in x)`$\}$. `dr1` is clearly more general than `dr2`. When considering the combination `(dr2, arg1)`, if $C_{a,p}$ has classifiers for both `(dr2,arg1)` and `(dr1, arg1)`, then both need to be updated with the training example, since if `p(dr2,arg1)` changes then so must `p(dr1,arg1)`. The update involves calling the learning procedure for the classifier ($learn$) and updating some measure of its reliability ($updateReliability$), to use during prediction to choose a winning prediction from multiple matching classifiers.

The set of classifiers $C_{a,p}$ also needs to be updated to include the combination of deictic terms just seen (*createNewClassifier*). For instance, suppose when considering the combination `(dr2, arg1)`, there was no matching classifier. If the default prediction of no change for `p(dr2, arg1)` is correct, there is no need to update $C_{a,p}$ because the prediction is already correct. Otherwise a classifier labelled with the `(dr2,arg1)` deictic term is created: it will be trained to predict when action a on its input state causes `p(dr2,arg1)` to change.

It might be that this deictic term is too specific, and then the preconditions can be learnt, but there will be several classifiers which all predict the same change for the same preconditions. To accommodate this possibility, whenever a new classifier is added, classifiers for more general deictic terms are also added. It is undesirable to add every possible more general deictic term combination, so only those supported by the data are added: namely the least general generalisations of the new classifier's deictic terms with every other classifier's deictic term. In the case of adding a new classifier for `(dr2,arg1)` when there was already a classifier for `(dr3,arg1)`, where `dr3` is $\{$`x:(at x arg1)` $\wedge$ `(in x)`$\}$ an additional classifier for the intersection case

```
Prediction:  Input:   Unlabelled instance $(x,a)$, model parameters $C_{a,p}$
             Output:  Prediction $\delta$

Prediction:
   for all $p \in P$ do
      $M := findArguments(x,p)$
      for all $m \in M$ do
         $y^m := getPrediction(C_{a,p}, x^m)$
   if $y^m = 1$ then
      $\delta = \delta \cup \{p(m)\}$

function $getPrediction(C, x^m)$
   $r := 0$
   for all $c \in C$ do
      if $deicticRefs(x^m) \supseteq deicticRefs(c)$ then
         if $r < reliability(c)$ then
            $y := predict(c,x)$
            $r := reliability(c)$
   return $y$
```

Figure 5.6: Outline prediction algorithm: each test example is a situation graph x and the action a whose effects are to be predicted.

({x:(at x arg1)}, arg1) = dr1 would be added, covering the possibility that the real change is to p(dr1,arg1) rather than to p(dr2,arg1). This new classifier initially is a copy of the (dr3,arg1) classifier, since all the training examples for (dr3,arg1) also apply to (dr1,arg1); however, it is then updated with the (dr2,arg1) training example, and in future training steps it will receive training examples containing all three combinations of deictic terms.

Conversely, the deictic term might be too general, and then many training examples this classifier sees will conflict, i.e., the same preconditions will sometimes be matched with a change to p(dr2,arg1) and sometimes with no change. This problem could be corrected for by considering the reliability of the predictions made by the classifier.

At prediction, given a test example x, again each predicate p is considered in turn, all combinations of possible arguments m in x are found and the same search for matching classifiers is performed. If no classifiers are found then the model predicts no change for the fluent $p(m)$. If exactly one classifier is found then its prediction is used, and if there are multiple matching classifiers (at different levels of generality), the classifier with the highest reliability score is used.

5.1.3 Classification

The implementation of the $learn$, $predict$, $updateReliability$ and $createNewClassifier$ procedures depends on the type of classifier used by the model. As before, a voted perceptron classifier is used. Thus the $learn$ and $predict$ procedures are implemented as previously described in Chapter 4.

For $updateReliability$, it is straightforward to create a simple measure of reliability for the voted perceptron classifier. Each classifier has a set of n support vectors s_i $(i = 1, \dots, n)$, and each s_i has an associated weight c_i. The value of c_i is initially set to 1 and is incremented every time the corresponding hypothesis makes a correct prediction. Therefore c_i is a count of the number of correct predictions by the corresponding hypothesis, plus 1. The number of support vectors, n, is the total number of incorrect predictions made during learning. The reliability of the classifier can then be calculated as the number of correct predictions $(\sum_{i=1}^{n} c_i - n)$ divided by the total number of predictions made $(\sum_{i=1}^{n} c_i)$. This reliability measure is used when there are several classifiers which could apply in the prediction of change to a particular relation: the classifier with the highest reliability score is used.

There is an implicit assumption that each classifier is more reliable than the *null classifier*, the classifier which predicts no change in any circumstance. Without noise the assumption holds, but when observations are noisy, some classifiers' positive training examples may just be noisy examples. In this situation, the null classifier should make better predictions on the same training examples. Therefore the *null reliability* — the reliability of the null classifier — is also calculated for each classifier. If the null reliability is higher than the classifier's reliability then, for the purposes of prediction, the classifier is replaced by the null classifier, and its reliability score is replaced by the null reliability. The null reliability is calculated as the number of training examples (seen by the classifier) which did not change, divided by the total number of predictions made.

Finally, *createNewClassifier* is called during learning when an example x is presented which does not exactly match any of the existing classifiers, *and* the default prediction of no change is wrong for the example. At this point *createNewClassifier* sets up a new voted perceptron with x as its first support vector, and labelled with the deictic terms present in x. Also, *createNewClassifier* is called when additional classifiers are needed to cover the deictic terms whose constraints are intersections between x's and existing classifiers' constraints. Each of these classifiers is initialised with all of the support vectors (and weights) from the parent classifier, since all examples which are covered by the parent classifier are also covered by the new classifier. The new classifiers are then updated with x, if the parent classifier misclassified x.

5.1.4 Measuring similarity between situation graphs

With the graph representation and structure of the learning model in place, it remains to identify a suitable kernel function, which each voted perceptron classifier uses to measure the similarity between situation graphs. The ideal kernel is one whose features are also situation graphs, corresponding to the conjunctive preconditions in the underlying rules. I define a family of kernels which operate on pairs of situation graphs G_1 and G_2. The fluent nodes of G_1 and G_2 are denoted R_1 and R_2 below. The kernel $K_{map}(G_1,G_2)$ counts the number of common subgraphs in G_1 and G_2, subject to a mapping restriction map. Each map $m \in map$ specifies a maximal mapping from the nodes in R_1 to nodes in R_2. Thus no $m \in map$ may be a subset of $m' \in map$ and each map corresponds to a single maximum common subgraph of G_1 and G_2.

Definition 5.1.3. The kernel $K_{map}()$ is defined as:

$$\begin{aligned}
K_{map}(G_1,G_2) &= \sum_{m \in map} 2^{same_m(G_1,G_2)}, \text{ where} \\
same_m(G_1,G_2) &= \sum_{r_1 \in R_1} \sum_{r_2 \in R_2} same_m(r_1,r_2), \\
same_m(r_1,r_2) &= 1 \text{ iff } label(r_1) = label(r_2), \text{ and} \\
&\quad \forall i (args_i(r_1), args_i(r_2)) \in m \\
same_m(r_1,r_2) &= 0 \text{ otherwise.}
\end{aligned}$$

If map contains all possible mappings between R_1 and R_2 then K_{map} is the subgraph kernel (Gärtner et al., 2003b). If map contains exactly one mapping between R_1 and R_2 then K_{map} is the DNF kernel (Sadohara, 2001; Khardon and Servedio, 2005). Computing the subgraph kernel is NP-hard (Gärtner et al., 2003b), while even the simplest case in this family of kernels, the DNF kernel, may scale poorly, since the PAC-learnability of DNF is an open problem. In light of the results in Chapter 4, I therefore additionally restrict the kernel family to $K_{map,k}()$ which counts common subgraphs with k or fewer fluent nodes, so that the kernels now range between the k-DNF kernel and graphlet kernels of size k (Shervashidze et al., 2009).

Definition 5.1.4. The kernel $K_{map,k}()$ is defined as:

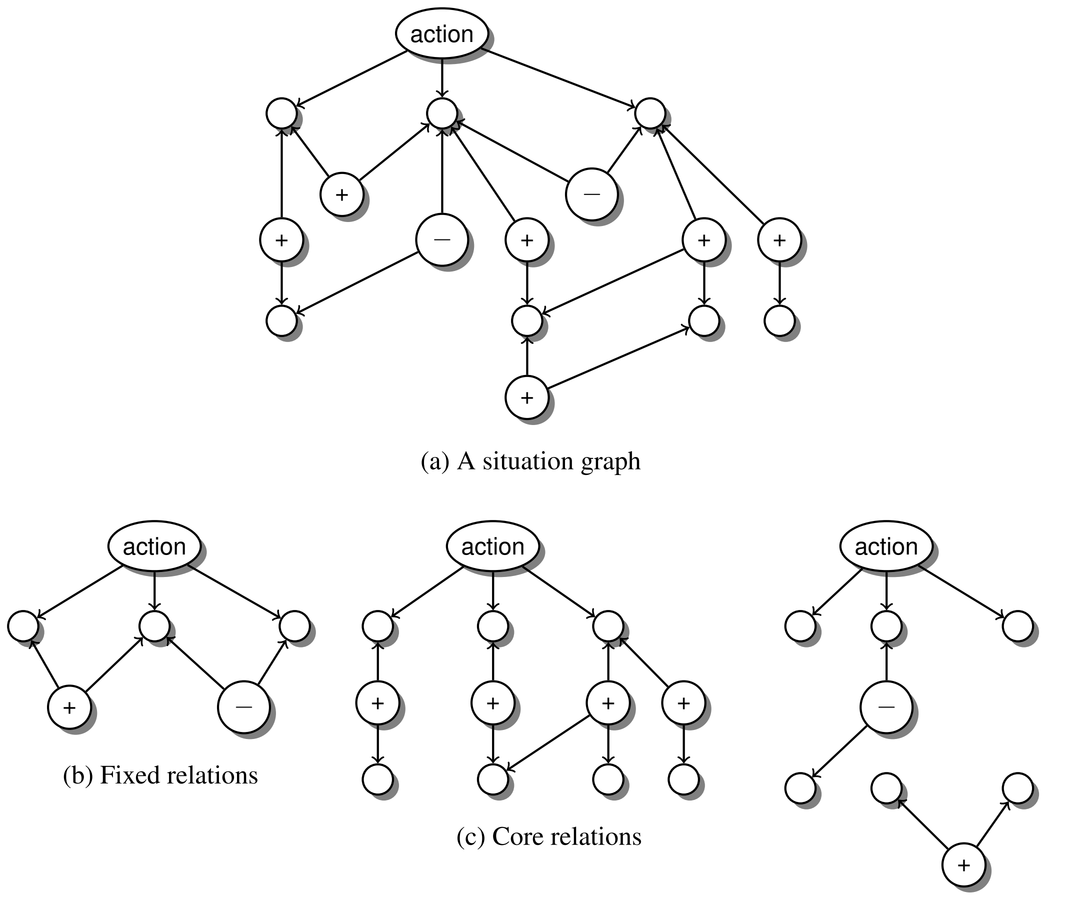
$$K_{map,k}(G_1, G_2) = \sum_{m \in map} \sum_{i=1}^{k} \binom{same_m(G_1, G_2)}{i},$$

where $same_m$ is as in Definition 5.1.3.

5.1.4.1 The deictic reference kernel

I select one member of our family of kernels, the *deictic reference kernel of size k*, whose particular map is one where object nodes must be mapped to nodes which have a common deictic term. The additional constraints on the deictic reference kernel allow an iterative procedure to be defined to calculate it. The procedure is based on the following observations. First, some nodes always have a fixed mapping: the i-th arguments of the action are always mapped to each other, as are the i-th arguments of the predicate the classifier is predicting. For example, if the kernel was being calculated for two Briefcase situation graphs, with a `move` action, and for the `(at [A] arg1)` classifier (see Figure 5.3c for a possible situation graph), then the $arg1$, $arg2$, and $[A]$ nodes would be fixed. Second, since the features of the kernel must also be situation graphs, the subgraphs it considers must also satisfy the positive link assumption (Section 5.1.1.1). Thus, any common subgraph considered by the kernel must contain a *core* subgraph with only positive fluents, linking every object node to the action parameters. Figure 5.7 shows an example of a situation graph split into fixed, core, and non-core relations. Furthermore, the number of positive fluents in a situation graph is typically small relative to the number of negative fluents, and so the set of all core subgraphs will typically be much smaller than the set of all subgraphs.

Fixed, core and non-core relations thus partition the set of possible subgraphs into those which contain purely fixed relations (fixed subgraphs), those which contain fixed and at least one core relation (core subgraphs), and those which also contain at least one non-core relation. By definition any subgraph in the last set must contain a core subgraph. The strategy is therefore to order the count of subgraphs so that fixed subgraphs are counted first (K_0 in Definition 5.1.5), followed by subgraphs containing core subgraphs, in order of increasing size of the core (K_n in Definition 5.1.5). We do so by first generating the set $core_i$ of possible core graphs with i positive fluents, where i is initally 1. Figure 5.9 gives an example of the sets of $core_i$ generated from two situation graphs.



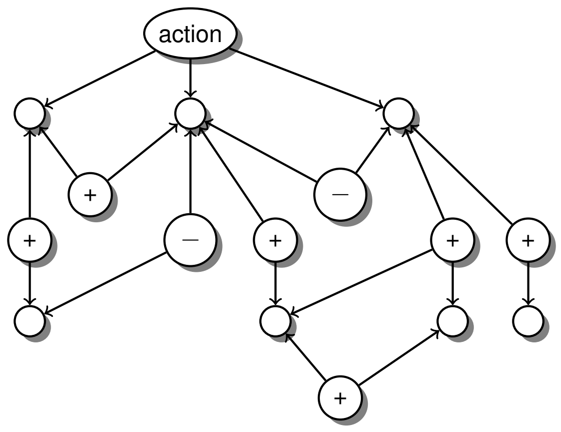
(a) A situation graph

(b) Fixed relations

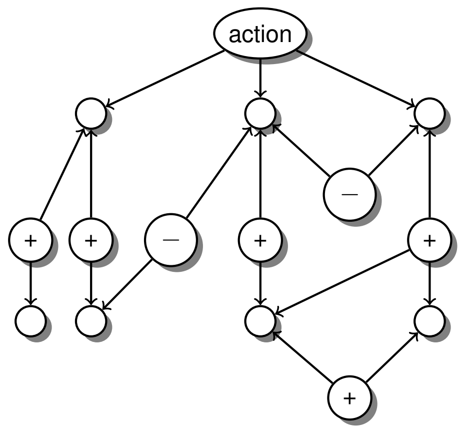
(c) Core relations

(d) Non-core relations

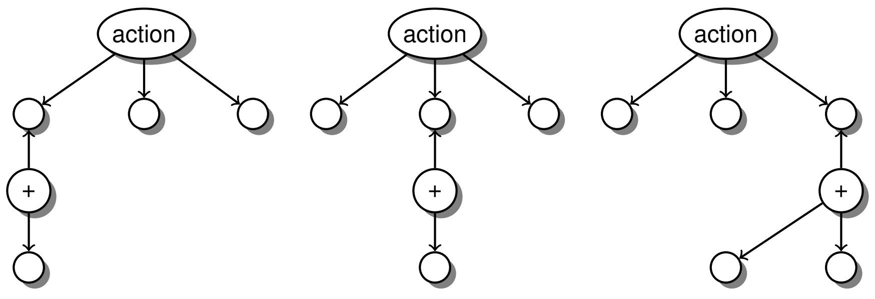
Figure 5.7: Fixed, core and non-core relations. In the situation graph in (a), true relations are denoted “+” and false relations “-”, while object nodes are empty. For the purposes of the example, all relations in the situation graph are assumed to have the same label. The sets of relations forming the fixed, core and non-core relations are shown in (b), (c) and (d), respectively.



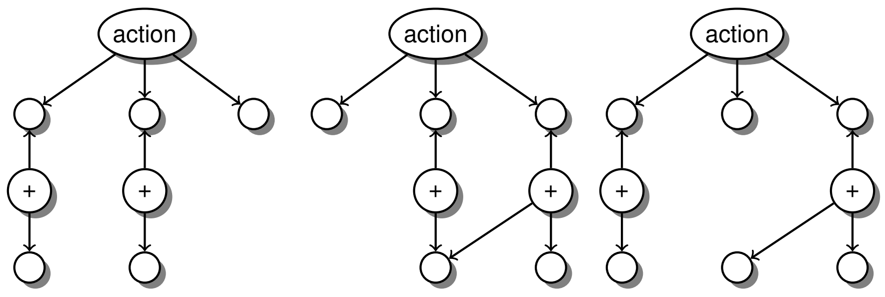
(a) G_1



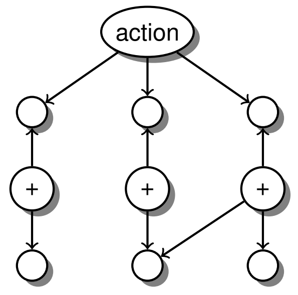
(b) G_2



(c) *core*$_1$



(d) *core*$_2$



(e) *core*$_3$

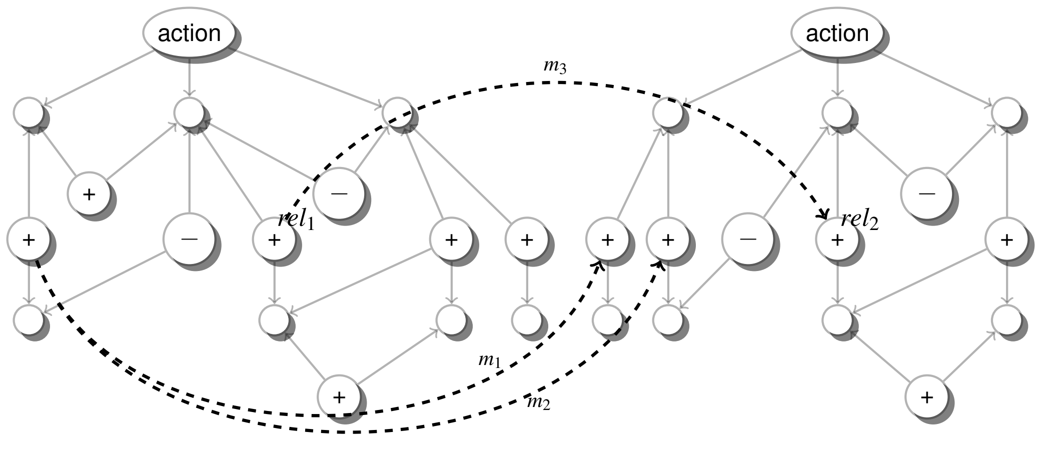
Figure 5.8: Situation graphs G_1 and G_2 with each set $core_i$ of subgraphs containing only i defining relations. Each set $core_i$ is calculated from the previous set $core_{i-1}$.

Then, for each such core graph, calculate the number of possible common situation subgraphs consisting of that core graph combined with any set of non-core fluents.[4] Each set $core_i$ is used to generate $core_{i+1}$, until there are no further possible common situation subgraphs of size $i+1$. By calculating the number of common situation graphs in this way, it is straightforward to limit the calculation to graphs with k fluents or fewer. In this case only core graphs of up to size k are ever considered. When combining any element of $core_i$ with non-core fluents, only combinations which have up to k fluent nodes are counted.

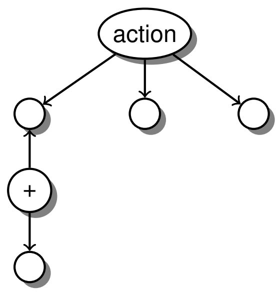
To calculate the number of common situation graphs with core $core_i$, we maintain for each $g \in core_i$ a set of mappings $maps_g$, each of which contains all maps from R_1 to R_2 which are consistent with g. When constructing $core_{i+1}$ from $core_i$, each $g \in core_i$ is extended by adding each possible core fluent $r_1 \in R_1$ (duplicates can be avoided by appropriate ordering of the fluents). At the same time $maps_g$ is extended by every pair of core fluents $(r_1 \in R_1, r_2 \in R_2)$, provided the resulting mapping of deictic references between G_1 and G_2 is injective. An example of the update to $core_i$ and $maps_g$ is shown in Figure 5.9.

We now use $maps_g$ to count the number of common situation graphs with core g. We first construct $otherrels$, the set of non-core relations in R_1 whose arguments are defined by g, that is, all the arguments of the relations have deictic references defined by fluents already in g. It remains to count (not generate) the fluents in $otherrels$ which have a matching relation in R_2, for each possible mapping in $maps_g$. Each $m \in maps_g$ defines a mapping from the subgraph g in G_1 to a subgraph h in G_2. A relation $r \in R_1$ matches $q \in R_2$ if they have the same label, and each argument of r is mapped by m to the corresponding argument of q. The number of matched relations under mapping m is denoted by $otherrels_m$. Then the number of situation subgraphs of size k generated by a particular positive subgraph g and mapping m is the number of ways of choosing k relations from the set of matched relations: $\binom{otherrels_m}{k}$. The full kernel calculation is formalised in the kernel definition (Definition 5.1.5).

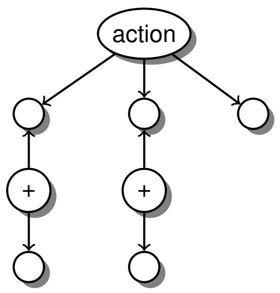
[4]Most non-core fluents will be negative fluents, but some positive fluents may be included, if their parameter list does not include any action parameters.



(a) Mappings of nodes sharing the same deictic term in G_1 (left) and G_2 (right). Nodes marked $+$ and $-$ are positive and negative fluents respectively, blank nodes are deictic terms. Mappings m_1 and m_2 map nodes which occur in sub-graphs g and h below, m_3 maps nodes only occurring in h.



(b) Subgraph $g \in core_1$: $maps_g = \{\{m_1\},\{m_2\}\}$



(c) Subgraph $h \in core_2$: $maps_h = \{\{m_1,m_3\},\{m_2,m_3\}\}$

Figure 5.9: An example of updating $core_i$ and $maps_g$. G_1 and G_2 are the situation graphs being compared by the kernel function. g and h are examples of graphs in $core_1$ and $core_2$ respectively. $maps_g$ contains mappings between G_1 and G_2 which define pairs of subgraphs which are isomorphic to g. When graph g is extended to h by adding a new core relation $rel_1 \in G_1$, we consider whether $maps_g$ is compatible with rel_1. First locate candidate matching relations in G_2, say rel_2, and construct the mapping of deictic terms between G_1 and G_2 given by mapping rel_1 to rel_2. Here that mapping is m_3. Now any mapping in $maps_g$ can be extended by adding m_3 to it, provided the mapping and m_3 do not conflict by mapping the same object in one graph to a different object in the other. Here this does not happen and so $maps_g$ is extended to $maps_h$ by adding m_3 to each mapping in $maps_g$.

Definition 5.1.5. The deictic reference kernel of size k, $K_k(G_1, G_2)$ is defined iteratively as follows:

$$K_k(G_1, G_2) = \sum_{i=0}^{k} K_{i,k}(G_1, G_2)$$

where $K_{i,k}$ is the number of common subgraphs of G_1 and G_2 with core of size i and containing up to k fluent nodes:

$$\begin{aligned}
K_{0,k}(G_1, G_2) &= \sum_{i=1}^{k} \binom{same_0(G_1, G_2)}{i}, \text{ where} \\
same_0(G_1, G_2) &= \sum_{r_1 \in R_1} \sum_{r_2 \in R_2} same_0(r_1, r_2), \\
same_0(r_1, r_2) &= 1 \text{ iff } label(r_1) = label(r_2) \\
&\quad \text{and } args(r_1) \subseteq args(a_1) \\
&\quad \text{and } args(r_2) \subseteq args(a_2) \\
&\quad \text{and } \forall i, j \; args_i(r_1) = args_j(a_1) \iff args_i(r_2) = args_j(a_2), \\
same_0(r_1, r_2) &= 0 \text{ otherwise.}
\end{aligned}$$

$$K_{n,k}(G_1, G_2) = \sum_{g \in core_n} \sum_{m \in maps_g} \sum_{i=0}^{k-n} \sum_{j=0}^{k-n-i} \binom{otherrels_m(G_1, G_2)}{i} \binom{same_0(G_1, G_2)}{j}$$

5.1.5 Complexity

The complexity of the learning algorithm depends on the number of training examples, the number of classifiers the algorithm generates and must update, and the complexity of the kernel calculation.

The kernel calculation is dominated by the calculation of $K_{n,k}(G_1, G_2)$, whose complexity depends on the number of core relations in each graph, here denoted R_1^+ and R_2^+, and the number of non-core relations in each graph, denoted R_1^- and R_2^-. The complexity is the product of the size of $core_i$, the size of $maps_g$ for any $g \in core_i$ and the complexity of calculating $otherrels_m$ for any $m \in maps_g$. The size of each $core_i$ set is determined by the number of relations in R_1^+: $|core_i| \leq |R_1^+|^i$. Whenever an element of $core_i$ is extended by adding a new relation $r_1 \in R_1^+$, the corresponding map g for that element is extended by adding each $r_2 \in R_2^+$ which is a valid match. Thus the size of $maps_g$ for each $g \in core_i$ is $O(|R_1^+|^i|R_2^+|^i)$. Finally, calculating $otherrels_m$ for each $m \in maps_g$ requires at most comparing every non-core relation in R_1^- with every non-core relation in R_2^- to check it is consistent with m. The number of comparisons is therefore $O(|R_1^-||R_2^-|)$ and the cost of comparison is a search through $maps_g$, $O(log(|R_1^+|^i|R_2^+|^i))$. The $overrels_m$ calculation therefore has complexity $O(|R_1^-||R_2^-|log(|R_1^+|^i|R_2^+|^i))$. Overall then the kernel calculation runs in $O(|R_1^+|^{2k}|R_2^+|^k|R_1^-||R_2^-|log(|R_1^+|^k|R_2^+|^k))$. With k set to 3 this is polynomial but may not be particularly efficient in the worst case, as the sets of non-defining relations R_1^- and R_2^- could be large relative to the number of objects in the situation graph.

The number of classifiers generated during training depends on the deictic terms encountered in the training set. Whenever a deictic term is encountered which does not have a matching classifier, not only is a classifier for the deictic term created, but also classifiers are created for any deictic terms with constraints which are intersections of the new deictic term constraint with a constraint of an existing classifier. This process can lead to an exponential growth in the number of classifiers. In practice this is mostly a problem when there are noisy observations, as many erroneous deictic terms are encountered. Essentially the same problem exists in prediction, where the resulting erroneous classifiers must be identified in order to prevent noisy predictions. As discussed in section 5.1.3, this can be resolved in prediction by comparing the reliability of each classifier to the null classifier, and only accepting prediction from the classifier if it is more reliable than the null classifier. A similar tactic could be adopted to periodically clear out unreliable classifiers during learning.

5.2 Experiments and results

In the experiments in this chapter, examples are taken from a real-world robot domain, and from simulations of standard planning domains used in the International Planning Competition, as described in Chapter 3. The robot domain is inherently partially observable and noisy, and so corresponds to a world-level observation model. The perceptual function, mapping sensor values to percepts, is hand-coded. In the case of the standard planning domains, a percept-level observation model is used: observations are simulated by generating fully-observable, noiseless descriptions of the world. The model was also trained on the same data, modified by applying a blocking process, to simulate noise and partial observability (Section 3.5).

5.2.1 Experiments in simulated planning domains

As before, the model was evaluated using data simulated from STRIPS domains (BlocksWorld, ZenoTravel, Depots, DriverLog and Rover) and in this case, also extended STRIPS domains (Briefcase, Elevator). The remaining parameters of the experiments remained the same as in Chapter 4.

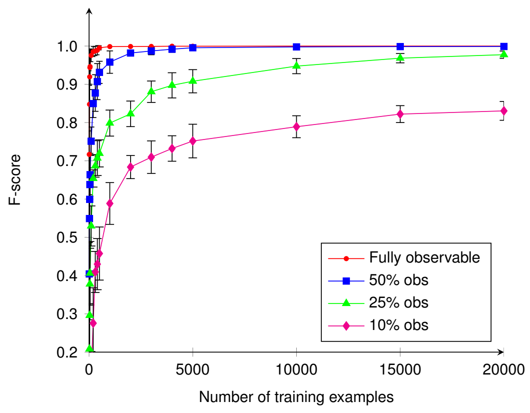
5.2.1.1 Results

Results of the experiments on noiseless domains are shown in Figures 5.10 and 5.11. The results are qualitatively similar to those for the vector representation. The fully observable cases are easily learnt and the F-score is 1 or very close to 1 after 1,000 training examples, with the exception of the Rovers domain; the extended STRIPS domains appear to be slightly more difficult to learn, which is to be expected, since the preconditions are more complex. In the partially observable cases, as expected, performance degrades with lower numbers of observed fluents, but the model is clearly producing reliable predictions which improve with increasing numbers of training examples.

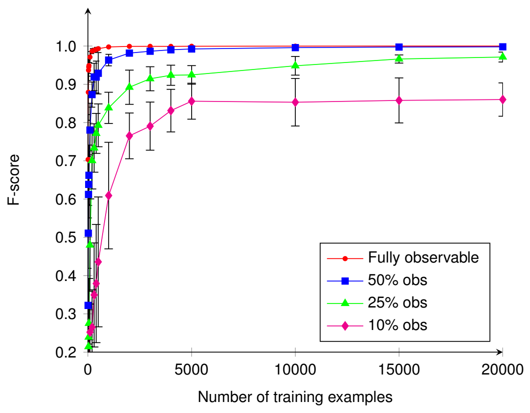
The results for Rovers are considerably worse for the graphical representation compared to the vector representation (Figure 4.7e). The only difference is that now first-order deictic terms are included in the learning process. As a result there are now many more possible hypotheses for each action precondition, to the extent that in Rovers the learning model overfits the data. A possible solution to this problem would be to learn using both zero-order and first-order deictic references and introduce a model selection step to choose the best model. Alternatively a bias towards simpler (i.e. zero-order based) hypotheses could be used, similar to the approach of Pasula et al. (2007).

Results of experiments in noisy domains are shown in Figures 5.12 and 5.13, in the latter, more difficult domains, only up to 10,000 training examples. Beyond 10,000 examples the algorithm starts to take longer times to run the 2,000 test cases ($>$ 12 hours) on these domains, as a consequence of the growing number of classifiers. As discussed in Section 5.1.5, this could be resolved by considering the reliability of the classifiers at intervals during learning. Alternatively, rule extraction (Chapter 5) could be used, if extended to run on the graphical representation: by repeatedly "compacting" the prediction function to a STRIPS-like rule from a kernel calculation using of the order of 1,000 support vectors, both training and prediction times would be significantly shortened.

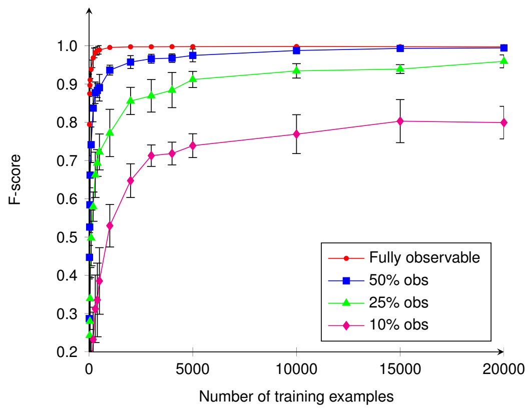
The results for the Rover domain were particularly adversely affected, and these results are not shown. The Elevator domain also proved to be problematic for the fully observable, noisy cases, to the extent that no results are available for this case. The problem lies in the highly connected and redundant nature of the Elevator domain: every floor is related to every other floor by the `above` predicate. The introduction of noise makes matters worse, as normally deictic references force a grouping of floors into those above or below various combinations of the action parameters, but noise establishes multiple different additional groupings. As a consequence, the space of possible preconditions and effects expands combinatorially. Since the algorithm is able to learn reasonable action models when examples are additionally incomplete, this is at least partly a problem with the coding of the domain, and is likely to affect other learning algorithms similarly.



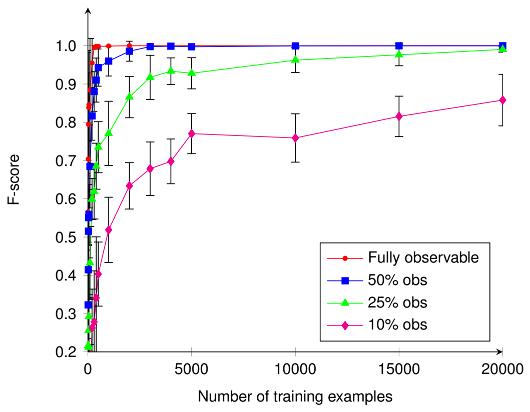
(a) Blocksworld



(b) ZenoTravel

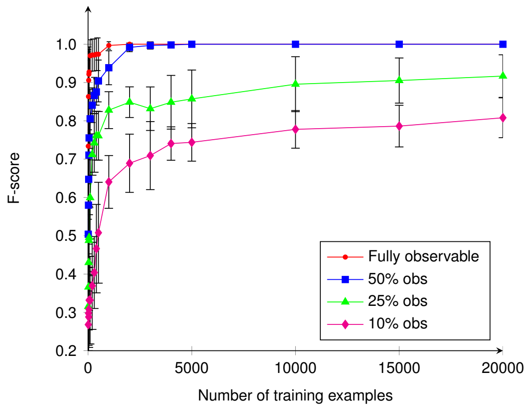


(c) Depots

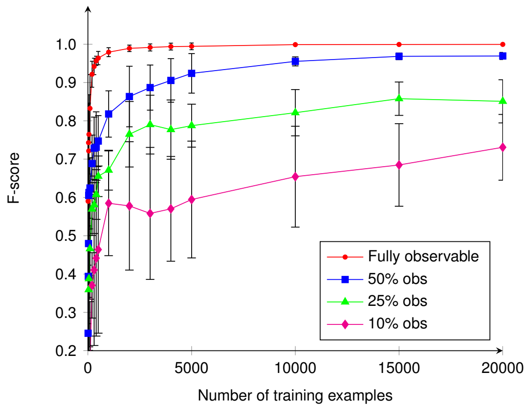


(d) DriverLog

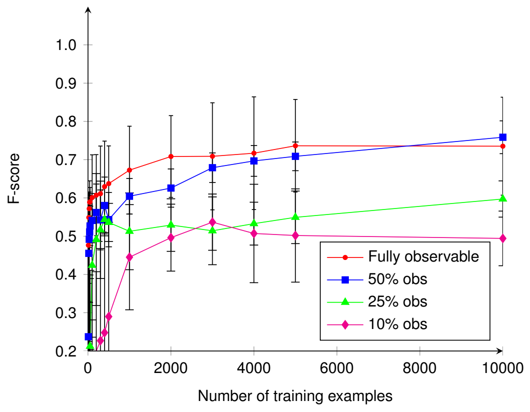
Figure 5.10: Results from learning actions in simulated planning domains, without noise, and at varying levels of observability: BlocksWorld, ZenoTravel, Depots and Driverlog.



(a) Briefcase

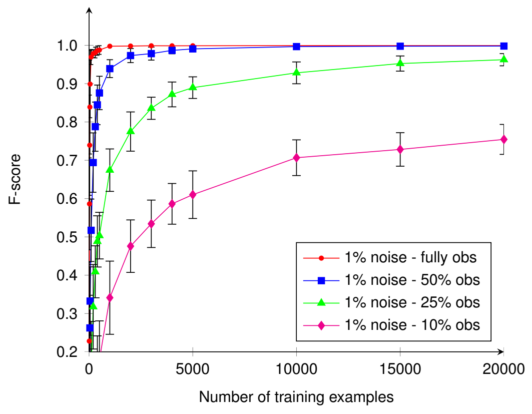


(b) Elevator

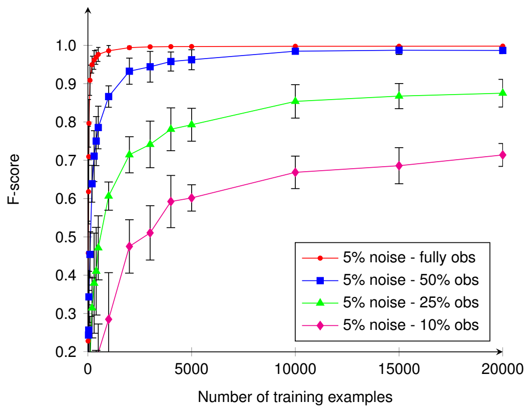


(c) Rover

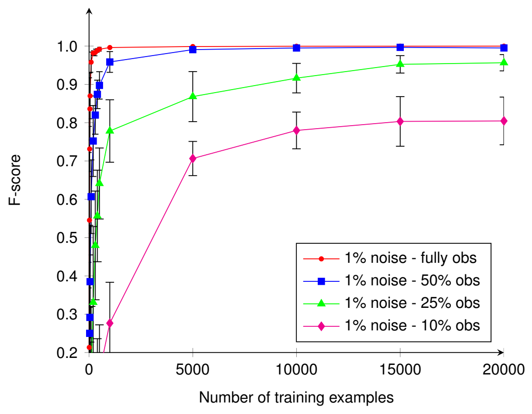
Figure 5.11: Results from learning actions in simulated planning domains, without noise, and at varying levels of observability: Briefcase, Elevator and Rover



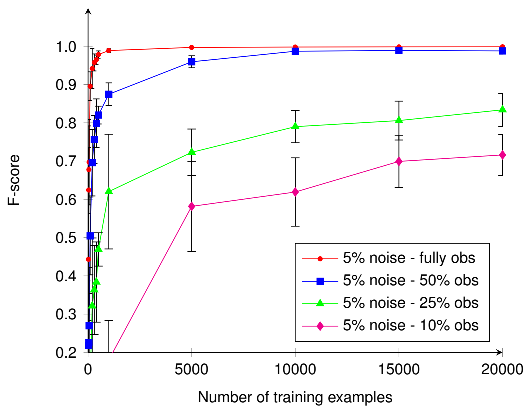
(a) Blocksworld: 1% noise



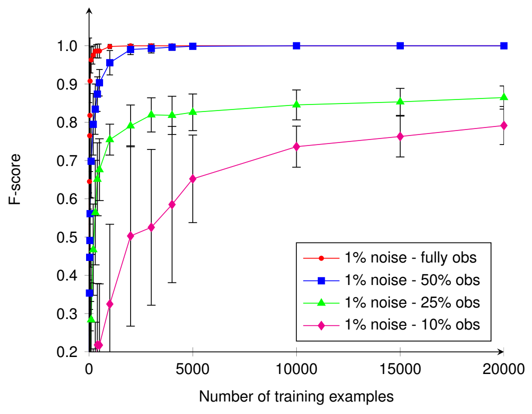
(b) Blocksworld: 5% noise



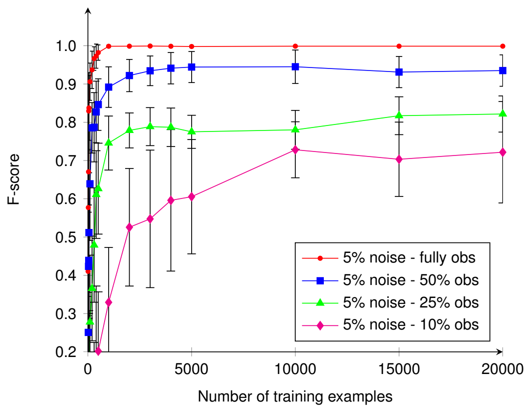
(c) Zeno: 1% noise



(d) Zeno: 5% noise

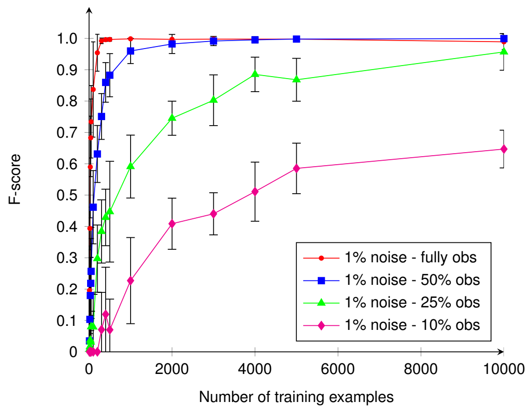


(e) Briefcase: 1% noise

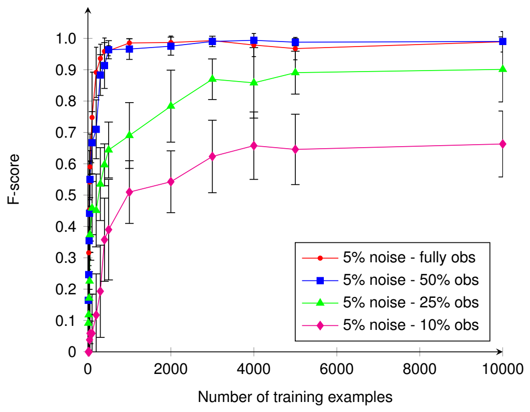


(f) Briefcase: 5% noise

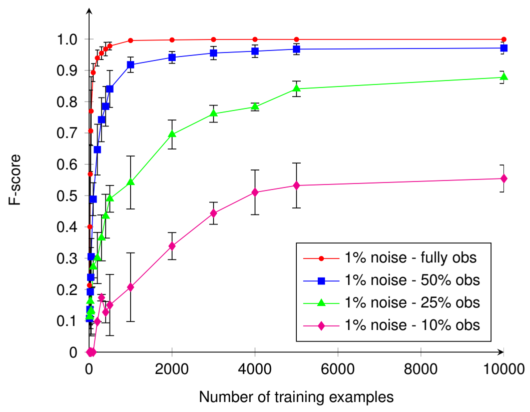
Figure 5.12: Results from learning actions in simulated planning domains, with varying levels of noise, and observability: BlocksWorld, Zeno, and Briefcase.



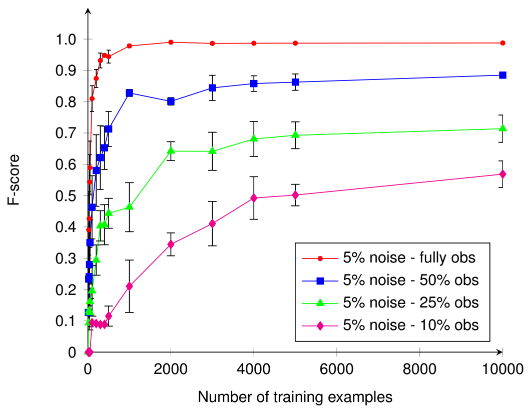
(a) Driverlog: 1% noise



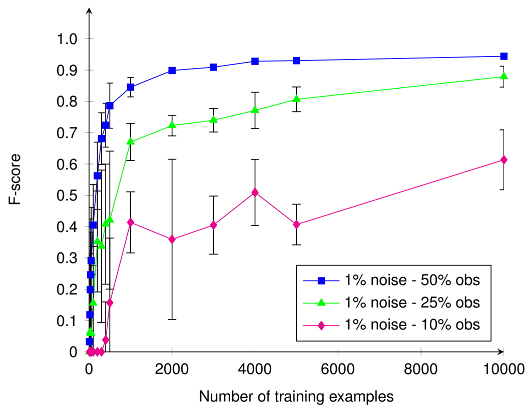
(b) Driverlog: 5% noise



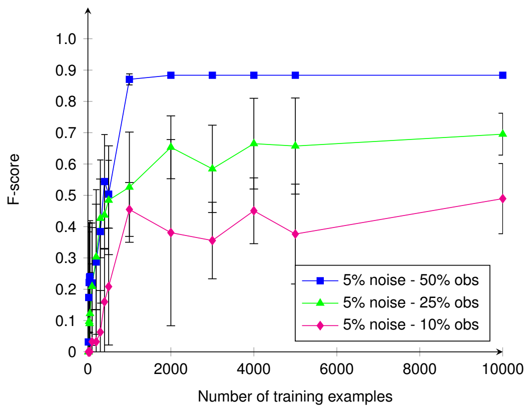
(c) Depots: 1% noise



(d) Depots: 5% noise



(e) Elevator: 1% noise

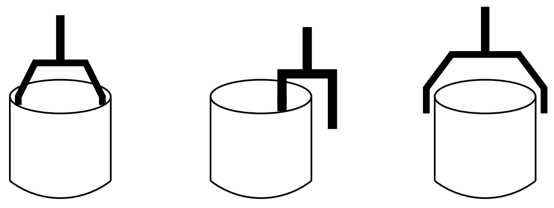


(f) Elevator: 5% noise

Figure 5.13: Results from learning actions in simulated planning domains, with varying levels of noise, and observability: Driverlog, Depots and Elevator.

5.2.2 Robot experiment

The set-up for the robot experiment was an industrial robot arm (Stäubli RX-60B) with 6 degrees of freedom, a two finger parallel gripper with haptic sensing (Schunk PG 70), and a Point Grey BumbleBee2 stereo camera. The robot's world consisted of a set of objects, each positioned on either a table or a shelf. The shelf space was restricted to hold a maximum of two objects. The objects were open or closed cylinders of different widths, heights and colours. The robot was able to grasp objects using three different grasps (see Figure 5.14), put objects on the table or the shelf, and sense whether an object is open or closed. The set of actions available to the robot, and the world and object properties are shown in Figure 5.15.



(a) Inner grasp (b) Edge grasp (c) Over grasp

Figure 5.14: Robot grasp types

The robot detected the locations and radii of objects using a circle detection algorithm applied to snapshots of the world taken by the camera (Başeski et al., 2009). The perceptual function, mapping sensor values to percepts, was hand-coded. Object properties such as `ontable` and `isin`, and the number of object slots available on the shelf, `shelfspace`, were automatically calculated at each observation, based on the circle detection data. The exceptions were `ingripper` and `gripperempty`, which were determined by haptic sensing; `open`, which was detected using a special sensing action `senseOpen`; and `knowWhetherOpen`, which indicates whether `senseOpen` was used on the object.

```
(define (domain robot-stacking)
  (:predicates (gripperempty)
    (ingripper ?x)
    (onshelf ?x)
    (ontable ?x)
    (clear ?x)
    (open ?x)
    (knowOpen ?x)
    (reachableInner ?x)
    (reachableEdge ?x)
    (reachableOver ?x)
    (aboveMinInnerRadius ?x)
    (belowMaxInnerRadius ?x)
    (aboveMinEdgeRadius ?x)
    (belowMaxOverRadius ?x)
    (isin ?x ?y)
    (instack ?x ?y))

  (:functions (shelfspace)
    (radius ?x))

(:action innerGrasp
  :parameters (?ob)
  :precondition  (and (reachableInner ?ob) (clear ?ob) (gripperempty) (ontable ?ob)
                      (aboveMinInnerRadius ?ob) (belowMaxInnerRadius ?ob))
  :effect (and (ingripper ?ob) (not (gripperempty)) (not (ontable ?ob))))

(:action edgeGrasp
  :parameters (?ob)
  :precondition (and (reachableEdge ?ob) (clear ?ob) (gripperempty) (ontable ?ob)
                     (aboveMinEdgeRadius ?ob))
  :effect (and (ingripper ?ob) (not (gripperempty)) (not (ontable ?ob))))

(:action overGrasp
  :parameters (?ob)
  :precondition (and (reachableOver ?ob) (gripperempty) (ontable ?ob)
                     (belowMaxOverRadius ?ob))
  :effect (and (ingripper ?ob) (not (gripperempty)) (not (ontable ?ob))))

(:action putAway
  :parameters (?ob)
  :precondition (and (ingripper ?ob) (shelfspace > 0))
  :effect (and (onshelf ?ob) (gripperempty) (not (ingripper ?ob)) (shelfspace=shelfspace-1)))

(:action putOntable
  :parameters (?ob)
  :precondition  (ingripper ?x)
  :effect (and (gripperempty) (ontable ?ob) (not (ingripper ?ob))))

(:action senseOpen
  :parameters (?ob)
  :precondition  (and (not (knowWhetherOpen ?ob)) (ontable ?ob))
  :effect (knowWhetherOpen ?ob)))
```

Figure 5.15: Representation of high-level actions in the object stacking domain (adapted from Petrick et al. (2010))

The robot's observations and actions were noisy. When detecting objects, both false positives and false negatives could occur. In experiments, 97.2% of objects present were detected, and 72.9% of detected objects were actually present in the scene (Başeski et al., 2009).[5] Additionally, the size and/or location of a detected object could be incorrect. This led to inaccuracies in the reachability calculations, and to objects being incorrectly observed as inside other objects and vice versa. It also meant actions could fail even when preconditions for the action appeared to be satisfied. For example, if the object location or size were incorrect, the calculated grasp coordinates for a grasping action might not match the actual position of the object, and so the action would fail.

Additionally, the grasping actions were constrained in different ways. A grasp action could only be performed on an object, firstly, if the gripper was empty, and secondly, if the object was reachable (given by `reachableInner`, `reachableEdge` and `reachableOver`), otherwise the robot performed a no-op. Also, some objects could not be grasped by some grasps. The inner grasp could not be used on closed objects, or on objects larger than the outer span of the gripper, or objects with radius smaller than the size of the gripper fingers. The edge grasp could not be used on closed objects, and the over grasp could not be used on objects larger than the inner span of the gripper. The robot was allowed to attempt to use any grasp on any object.

The putting actions were also constrained. If the gripper was empty, or if the action was `putAway` and `shelfspace` = 2, then a no-op was performed. Similarly, because `senseOpen` was implemented by attempting to position the closed gripper inside an object, a no-op was performed if the gripper was not empty.

The learning model was used to learn preconditions and effects of actions in the robot domain (restricted to `innerGrasp`, `edgeGrasp`, `overGrasp`, `putDown`, `putAway`, and `senseOpen`). An action-observation trace of the robot's domain model was obtained by performing a random walk through the state space. While generating training data, the robot's world consisted of the table, shelf and four cylindrical objects distributed around locations on the shelf and table. The four objects were selected

[5]This pattern of noise is somewhat different to that seen in the blocking process model. Here, the accuracy of the fluents mostly depends on whether the object(s) involved were correctly detected. Since this largely depends on light conditions, objects in stacks are typically more error-prone, and so fluents relating to such objects, especially `isin` and `instack` fluents, will be noisier than other fluents. Also size and reachability fluents are more likely to be wrong than `ontable` and `onshelf`, due to differences in the granularity of measurements required. Since much noise is due to light conditions, repeated observations could help to improve accuracy, although there is a limit to the extent to which light conditions can be expected to change in a short timescale.

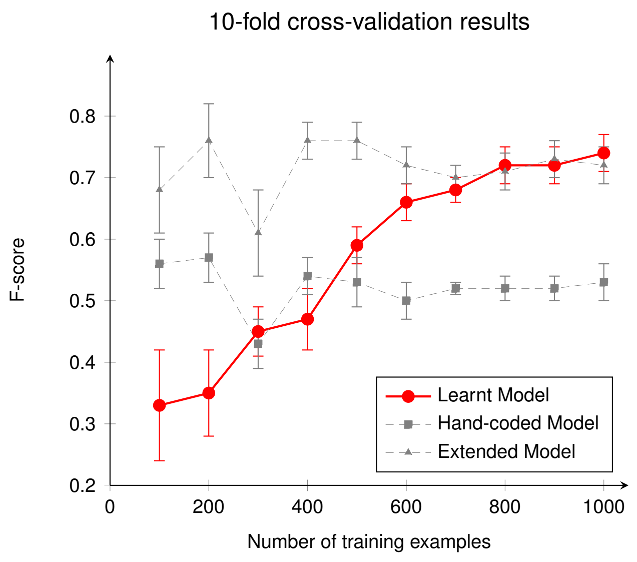


Figure 5.16: Results from the robot experiment: Mean F-scores from ten-fold cross-validation for the learnt model, in comparison to mean F-scores for predictions from the original hand-coded model, and an extended hand-coded model.

from a pool of objects of varying radii, depths and visual properties. Objects in the world were occasionally swapped, by an experimenter, for another object from the pool with different properties. In 40% of trials, some objects were turned upside-down to act as closed objects.

At each step, the robot made an observation of the world state, performed a randomly chosen action, and then made a new observation of the world. All actions attempted by the robot were included in the sequence of action-observation steps, regardless of whether the action was successful or not. The robot performed 1,000 action-observation steps in total over the course of a series of separate experiments, with different sets of four objects, and then a model was learnt from the data.

5.2.2.1 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 balanced F-measure, 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 a baseline model: the hand-coded domain description which had been used to define the behaviour of the robot (see Figure 5.15). For more than 500 training examples, the learnt model performs significantly better than the hand-coded domain model ($p < 0.001$). For example, with 1,000 training examples, the learnt model has a mean F-score of 0.74, in contrast to a mean F-score of 0.53 for the hand-coded domain. The superior performance of the learnt model suggested that some regularities in the domain were not captured by the hand-coded model. Therefore the hand-coded model was extended with as many additional preconditions and effects as could be identified, namely:

- extra conditional effects for each grasp action, to indicate that any reachable predicate which was true before a grasp would be false after a grasp: `(∀?x).(reachableInner ?x)` ⇒ `(not(reachableInner ?x))` and analogously for `reachableEdge` and `reachableOver`;
- corresponding extra effects for `(putAway ?x)` and `(putOnTable ?x)` to indicate that objects which have been put down will be reachable (although not always true, it is often the case): `(reachableInner ?x)`, `(reachableEdge ?x)` and `(reachableOver ?x)`;
- the addition of `gripperempty` as an extra precondition for `senseOpen`, as a consequence of the implementation of `senseOpen` on the robot;
- removal of all effects of `graspOver`, converting it into a no-op, since `graspOver` usually fails in the training data.[6]

For 600 or more training examples, there is no significant difference between the mean F-scores of the extended model and those of the learnt model ($p > 0.05$).

5.3 Discussion

The results show that the deictic reference kernel of size k is a suitable choice for learning action models in both STRIPS domains and domains extended to include conditional and quantified effects. As with the k-DNF kernel, the combination of the

[6]Such an action would be better learnt using a probabilistic representation of the action rules (e.g. as discussed in Section 6.5.3) rather than as here, where only the most probable result is learnt. Alternatively, Mugan (2010) gives methods for learning progressively more reliable actions, even from very low rates of success, in a continuous robotic world.

deictic reference kernel of size *k* with voted perceptrons allows learning to take place in noisy, partially observable domains. In contrast, using the full deictic reference kernel, which is a form of subgraph kernel, would have been computationally intractable.

Although most of the experiments were run on simulated domains, the results from the robot stacking domain give initial indications that the learning model will also handle noise and partial observability occurring in more realistic scenarios. The actions learnt in the robot domain did not include relational preconditions or effects: as such it is only a limited test of the learning approach. Nevertheless, it learnt a good model of the action dynamics which improved upon the original hand-coded specification of the domain, despite incomplete, noisy observations and noisy actions.

5.3.1 Relation between vector and graph representations

The graphical representation is a generalisation of the vector representation used in Chapter 4. The connections between the learning algorithms used for the vector and graph representations are discussed below, by considering how to convert from the vector-based algorithm to the graph-based one.

Any vector state description can be written as a situation graph, by putting the action and its arguments as nodes in the graph, and then adding nodes (with corresponding edges) for each fluent which has a 1 or -1 entry in the vector. In this graph, the deictic references are restricted purely to the action parameters, and so the resulting graph remains a STRIPS description (and therefore only has 0-order deictic references). However, it has the advantage of being a sparser representation than the vector, since unobserved fluents are still assigned an entry in the vector, but not assigned a node in the graph.

The number of classifiers used in either the vector or graph representations is the same. In principle, there is a classifier for each entry in the vector, or equivalently, each possible node in the situation graph. The classifiers are only activated as required, based on the results of the kernel calculation. Since the kernel calculation in both vectors and graphs is identical for a STRIPS domain (see below), the set of classifiers used is also identical.

The kernel used by the vector representation is the k-DNF kernel, whereas the graph representation uses the deictic reference kernel of size *k*. However, for two situation graphs G_1 and G_2 which only have 0-order deictic references, there is only one possible mapping between the nodes of the two graphs: the action parameters must be mapped to each other, and as in this case fluents only relate action parameters, each fluent node in one graph has either one or no corresponding fluent node in the other graph. In terms of the kernel calculation described in Section 5.1.4.1 and Definition 5.1.5, there are no core relations and so the calculation of $K_k(G_1, G_2)$ simplifies to just $K_{0,k}(G_1, G_2)$, that is, the number of common situation graphs is $\sum_{i=0}^{k} \binom{same_0(G_1,G_2)}{i}$, where $same_0(G_1, G_2)$ is the number of fixed fluent nodes common to G_1 and G_2. This is exactly the calculation performed by the k-DNF kernel. The only difference is that in the vector representation, common fluents are identified by comparing values of matching entries in two vectors, whereas in the graph representation, Definition 5.1.5 suggests a pairwise comparison of all pairs of nodes in the graph in order to identify matching fluents. Clearly this could be made more efficient by using a fixed-order graph traversal, which would then have the same complexity as traversing a vector.

The learning algorithm for the graphical representation is thus identical to that of the vector representation, *for STRIPS domains*, and with fixed-order graph traversal the computational complexity will also be the same. Once the situation graphs use higher-order deictic references, the calculation becomes more complex. In particular, fixed-order graph traversal can no longer be used to match common fluents in the two graphs, because any fluent in one graph may have multiple matching fluents in the other graph. It is this difference which requires the graph-based learning algorithm to take a subgraph isomorphism approach when matching graphs.

5.3.2 Related work

Since this chapter extends the approach in Chapter 4, much of the discussion on previous work in Chapter 4 also applies here. Further comment can be made on the use of deictic references, which have previously been used to reduce the size of the rule space and provide generalisation capabilities (Agre and Chapman, 1987; Finney et al., 2002; Pasula et al., 2007; Rodrigues et al., 2010a; Xu and Laird, 2010).[7] However, in addition, I use deictic references to compare states, by matching objects which share deictic terms. Some approaches have used the simplest form of deictic reference matching, matching only action parameters across states (Halbritter and Geibel, 2007). This is only suitable for STRIPS domains where it is assumed that all objects which have a role in the action are listed in the action parameters. The vector representa-

[7] Although not explicitly using deictic reference, the proximity heuristic used by Xu and Laird (2010) has strong similarities.

tion in Chapter 4 also included the assumption that objects with the same role would fall in the same position in their vector state representation; these objects were then matched for state comparisons. This is a simplified form of deictic reference matching, since under partial observability or noise the position assumption can fail (but will only strongly impact learning in non-STRIPS domains). The approach in this chapter is a generalisation to full deictic reference matching, allowing learning in domains affected by noise and partial observability. It also permits learning in domains where the assumption no longer applies that objects in the preconditions or effects are listed in the action parameters — namely domains such as Briefcase and Elevator.

5.3.3 Limitations and possible extensions

5.3.3.1 Extending beyond first-order deictic references

The state representation used in this chapter restricts the set of objects in the state to those listed in the arguments of the action, or directly related to those arguments (one step away from the action). This accordingly limits the action models which can be learnt to those whose preconditions and effects only contain the restricted set of objects. In many domains, such as the experimental domains used in this chapter, the restriction to first-order deictic references is sufficient to learn the domain, but learning in more complex domains may be affected.

Simply extending the set of objects to more distantly related objects is possible, but each additional object extends the hypothesis search space and will increase the computation time. One solution may be to learn or derive new predicates, which form the direct relation needed between an object and the action parameters. For example, when moving a tower of blocks, an `instack` relation might be required to relate each block to the bottom block which is the only block being acted upon. Some predicate invention of this nature has been carried out by Pasula et al. (2007).

A wider-reaching solution might be to proceed in a recursive manner, by allowing effects to also include actions. Then the effect of an action can result in another action, whose arguments are now the objects which were previously only related to the action arguments. In this way a chain of effects could be specified. For instance, in the Briefcase domain, learning the `move` action involves learning that objects inside the briefcase also move. This change is specified in terms of changes to the locations of those objects, but could be specified as a `move` action now performed on each object inside the moved object. However, it is not clear how to learn that certain indirect effects correspond to actions, and such a representation is also quite different from standard planning representations.

5.3.3.2 Alternative kernel calculations

Although the deictic reference kernel of size k is more efficient than the subgraph kernel, it is still quite expensive to compute. There may be scope to use an alternative graph kernel instead. The main issues to consider are the efficiency of the graph kernel, and its expressivity: the extent to which it distinguishes between graphs representing states with different outcomes, and identifies as similar graphs representing states with the same outcome.

The deictic reference kernel described in this chapter is essentially the k-graphlet kernel (Shervashidze et al., 2009), restricted so that the graphlets (or motifs) may only match where the deictic terms also match. Shervashidze et al. (2009) improve the speed of the kernel calculation by taking a sampling approach, which could also be considered here. Of particular interest is that they additionally give an algorithm to efficiently count graphlets up to size 5, where the graphs are of bounded degree, as is the case for the relation nodes in situation graphs. A modification to this algorithm to account for deictic references could potentially produce efficiency gains.

Other graph kernels are often based on walks and paths in graphs (Ramon and Gärtner, 2003; Borgwardt and Kriegel, 2005). A walk in a graph G is a sequence of edges $\langle(v_1,v_2),(v_2,v_3),\ldots,(v_{k-1},v_k)\rangle$ where each $(v_i,v_{i+1}) \in \mathcal{E}(G)$, while a path is a walk with no repeated vertices. An edge path is a walk with no repeated edges (but vertices may be repeated). Gärtner et al. (2003a) define two kernels which count matching walks in the two graphs being compared, where walks match either because they start and end on vertices with the same pair of labels, or because they share the same sequence of labels. Kashima et al. (2003) define a related kernel whose features correspond to the probability a particular label sequence is generated by a random walk in both graphs. These kernels are expensive (although tractable) to compute.

A more expressive kernel is the subtree pattern kernel (Ramon and Gärtner, 2003; Shervashidze and Borgwardt, 2009). A subtree pattern can be defined inductively as follows. If $G(V,E)$ is a graph then every $v \in V$ is a subtree pattern. If $t_1,\ldots t_n$ are subtree patterns, each rooted at r_i (where $r_i \neq r_j\ \forall i,j$) and $(v,r_i) \in E\ \forall i$ then $v(t_1,\ldots,t_n)$ is a subtree pattern. Note that this definition allows vertices to be repeated in subtree patterns, so that in the same way as walks extend paths by allowing repeated vertices, subtree patterns extend subtrees by allowing repetitions. Additionally this means that walks are special cases of subtree patterns. The original subtree kernel algorithm was computationally expensive, but recent work has developed a significantly more efficient algorithm (Shervashidze and Borgwardt, 2009), which makes it more efficient to calculate than the walk and path kernels discussed above.

The hypergraph kernel (Wachman and Khardon, 2007) is also based on walks, this time in hypergraphs, and uses *walk types* as the basis of the kernel. A walk in a hypergraph is defined to be a sequence of hyperedges, where each pair of consecutive edges share at least one node, and no consecutive edges are equal. The common node between each pair of edges must be specified. Hypergraph walks can be represented as strings of the form $P = p_1 i_1 j_1 p_2 i_2 j_2 \dots i_{n-1} j_{n-1} p_n$ where each p_k is a hyperedge, each i_k is the exit position from p_k and j_k the entry position into p_{k+1}. For example, a hypergraph representing a state in the ZenoTravel domain is

$\{$`(board person aircraft city)`,`(at person city)`,`(at aircraft city)`$\}$,

and one particular walk in this hypergraph is described by the string

`(board person aircraft city)`,1,1,`(at person city)`,2,2,`(at aircraft city)`.

A walk type is a generalisation of a walk, where each edge is represented only by its label, so the walk type of the walk above is $(\texttt{board}, 1, 1, \texttt{at}, 2, 2, \texttt{at})$. The hypergraph kernel $K_n(H_1, H_2)$ then counts the number of common walk types of some fixed length n in the two hypergraphs H_1 and H_2.

Could these kernels be used to learn action models instead of the deictic reference kernel? In terms of efficiency, the subtree kernel is best, with the most recent implementation running in time linear in the number of edges in the graph, and the maximum height of the subtrees considered. There can be many more edges in a situation graph than there are nodes, for example, in densely connected domains such as the elevator domain. The number of edges in a situation graph is bounded by the product of the number of *possible* fluents and the maximum arity of any predicate in the domain, m. The number of possible fluents is bounded by $|\mathcal{P}|\binom{|objs|}{m}$ where $|objs|$ is the number of objects from the original state represented in the situation graph. Thus the number of edges is bounded by $O(|objs|^m)$ and for densely connected graphs even the subtree kernel will be expensive to calculate.

A further issue is that without changes, all of the kernels could include features composed entirely of negative deictic terms (e.g., “the block which is not under the block I am unstacking and not on the floor”). This would lead to the kernels assigning high similarity scores to situations which are only similar in what they are not. For instance, where in one situation a block is on the table, and in another a block is stuck to the ceiling, the similarity calculation would be based on the fact that neither block is on the floor, or on the shelf, or in the bag. This measure of similarity seems counter-intuitive, and the deictic reference kernel was explicitly constructed to avoid it. Thus using any of the alternative kernels would involve adjusting the calculation to exclude cases of this form.

5.4 Summary

In this chapter the approach in Chapter 4 was generalised so that it could be applied to extended STRIPS domains. The generalisation relies heavily on deictic reference. It involved changing the representation of state observations from vectors to graphs by identifying that the vectors implicitly coded deictic references, and making this explicit in the graphical representation. The structure of the classification model was generalised so that classifiers were explicitly associated with deictic terms. Finally, a new graph kernel was defined, the deictic reference kernel of size k, which measures similarities between situation graphs by matching deictic terms.

The result is an incremental learning algorithm which can learn action models in both STRIPS and extended STRIPS domains even when observations are noisy and partially observable. Experimental results show that the generalised approach still performs well in STRIPS domains and also produces good action models in extended STRIPS domains. An approach which handles both noisy and incomplete observations is in itself novel in action model learning. In addition, the approach assumes only a weak domain model where predicates, action labels, and argument numbers and types are known, whereas many alternative approaches assume the availability of successful plans or the presence of a teacher.

Chapter 6

Extracting rules

A major limitation of the classification-based approach to learning the dynamics of a domain is that explicit rules are not generated by the model. Instead, it must be used as a black-box to make predictions of state changes, given an action and initial state. While such models can be used in model-based reinforcement learning (Sutton, 1990; Halbritter and Geibel, 2007), most planners require PDDL-style rules. Moreover, supporting a perceptron-based model of the world has practical problems relating to the storage of ever-increasing numbers of support vectors, and a prediction calculation which takes time proportional in the number of support vectors. Rules resolve the problem by providing a compact representation of the perceptron models. Finally, post-processing the perceptron models by extracting rules provides an opportunity to further filter out the effects of noise in the training data. In this chapter, I present a method to extract rules from classifiers trained on the vector-based state representation of Chapter 4, and discuss how it could be extended to work with the graph-based state representations of Chapter 5.

6.1 Existing approaches to rule extraction

The related problems of rule extraction from neural networks and from SVM classifiers have been well explored in the literature (Tickle et al., 2000; Martens et al., 2008; Barakat and Bradley, 2010). Many of these techniques could also be applied to extract rules from the voted perceptron model. A distinction which will prove useful is given by Tickle et al. (2000), who define the *translucency* of a rule extraction technique (for neural nets) to describe whether it operates on the internal weights and structure of the network (*decompositional*), or only on the external inputs and outputs (*pedagogical*).

Any pedagogical technique is immediately transferable between neural nets, SVMs and voted perceptrons, and while decompositional techniques are unlikely to apply directly, the underlying intuition may be reusable.

Tickle et al. (2000) also discuss other dimensions along which rule extraction techniques may be categorised:

- language: whether rules are generated in the form of symbolic Boolean, propositional, or relational rules; or as fuzzy rules;
- quality: the accuracy of each rule relative to the domain, and relative to the network from which it was extracted; the consistency of the rules extracted over different training sessions of the network; comprehensibility;
- computational complexity;
- portability to other networks.

In the type of rule extraction required here, rules should be generated as symbolic propositional or relational rules, in order to ultimately form PDDL rules (or similar) which could be used by a planning system. The intention is to extract rules which are as close as possible to those learnt by the perceptrons, with the assumption that the models will have learnt a reasonable approximation of the world action dynamics. It is important that the rules are consistent across different training sessions, as the classifiers for different effects could ultimately be trained with different sets of examples (e.g. under partial observability). If rules are to be combined across different effects to produce full action rules, then the rules for different effects of the same action-precondition pair need to coincide as much as possible. Clearly, computational complexity should be minimised, while portability to other networks is of little concern. The main choice, then, is whether to take a decompositional or a pedagogical approach.

6.1.1 Pedagogical approaches

In the pedagogical approach, rules are derived based on the inputs and outputs of the model, that is, based on the action, initial state and successor state tuples. This is almost the same problem as learning the action dynamics from world observations, but is significantly simplified, because, (i) observations can be selected as required, for instance, to be fully observed, and, (ii) noise is eliminated, as the observations will always be consistent with the action dynamics learnt by the model. As a consequence, existing techniques for learning planning operators, which require full observability and/or noiseless examples, could be used to learn the rules describing the perceptron's model.

There are some disadvantages in taking this approach. The resulting rules can only be as expressive as the two learning models combined. In particular, if learning in a noisy environment, the perceptron model is likely to produce many rules with conditional effects,[1] which most other approaches do not handle, so these would be lost or simplified in some way (not necessarily beneficially). Similarly, errors in the perceptron model could be magnified by the addition of a second learning process.

6.1.2 Decompositional approaches

In the decompositional approach, rules are derived using information internal to the model: the weights, biases, activation function and connections between nodes in a neural network, or the support vectors and decision function of an SVM. Additionally, the training data or model predictions may be used (also known as a *hybrid* or *eclectic* approach).

Decompositional SVM rule extraction methods are typically classed in terms of which aspects of the SVM model they use: the support vectors, the support vectors and decision function, or the support vectors, decision function and training data (Barakat and Bradley, 2010). However, it may be more useful to consider whether approaches are *data-driven* (they aim to define regions of the feature space which cover positive or negative training examples) or *feature-driven* (they aim to identify features which discriminate between the classes).

Data-driven approaches build candidate regions of the input feature space which are likely to contain mostly points of one class. The regions are adjusted to better fit the training data, according to some criteria, and then translated into rules. The initial creation of candidate regions may be achieved by clustering (of support vectors or training data) to find prototype vectors for each class (Zhang et al., 2005b; Núñez et al., 2008), and then using the prototypes, possibly in combination with the support vectors, to define regions. Alternatively, for certain kernels, initial candidate regions

[1]With noise, the perceptron model will occasionally see training examples with a spurious effect. The perceptron corresponding to the effect will usually learn that in general the effect does not occur, as it will be presented with many examples where the effect is not present. However, test examples very similar to those examples with the spurious effect may sometimes trigger a positive prediction from the perceptron. This will manifest itself in rule extraction as a conditional effect.

can be hyperrectangles constructed by projecting lines parallel to each axis from each support vector to the decision function (for non-linear RBF kernels), or from each axis to the decision function (for linear classifiers) (Fu et al., 2004; Fung et al., 2005). The data-driven approaches essentially approximate the SVM decision function with partitions of the input space which are easier to translate into rules. Most produce rules with low comprehensibility, as either all features are included in the rule antecedents (Zhang et al., 2005b; Núñez et al., 2008; Fu et al., 2004) or large numbers of rules are produced (Fung et al., 2005). This type of approach is therefore unsuitable for extracting rules from the voted perceptron models.

Alternative, feature-driven approaches, aim to identify which input features and values most contribute to the class prediction, and use only those features in classification rules. For instance, Barakat and Bradley (2007) determine the set of most discriminative features in the support vectors, by comparing the mean values of each feature in the positive and negative support vectors, and selecting those which give the best true positive to false positive rate over the full set of support vectors. At each stage in a sequential covering process, they refine a candidate rule by adding features which are increasingly less discriminative, until the addition of a feature does not improve on the rule prediction. At this point the rule is added to the rule set, all support vectors covered by it are removed, and the process repeats. Chen et al. (2007) select relevant features as part of the SVM training process, before generating hypercubes, in a similar fashion to the data-driven approach of Fung et al. (2005). Another method, specific to the families of DNF and polynomial kernels, is to attempt to identify the most discriminative features by seeking the highest weighted features in the feature space, and then conjoining those features to form rules (Zhang et al., 2004, 2005a). These approaches tend to produce rules with more compact rule antecedents, but comprehensibility can still be affected by the large number of rules produced, as noted by Barakat and Bradley (2010).

The principle behind decompositional neural network rule extraction methods is very similar to the feature-driven SVM extraction methods: find inputs which activate a neuron, by finding combinations of input weights which sum to more than the neuron's activation threshold. In the *subset method* (Fu, 1991; Towell and Shavlik, 1993; Krishnan et al., 1999), subsets of inputs which are guaranteed to fire a neuron are found. This leads to rules of the form "if the antecedent is true, then . . . ", where the antecedent is in DNF: a disjunction of the conjunctions formed by each subset. Since the computational cost of finding all subsets which fire a neuron is proportional to 2^n, where n is the number of inputs to a node, subset methods use various heuristics and constraints, such as limiting the number of antecedents or number of rules. A related method is the *M-of-N method* (Towell and Shavlik, 1993; Setiono, 2000) which extracts rules of the form “if M of the following N antecedents are true, then…”. The key observation underlying the M-of-N method is that groups of antecedents form equivalence classes where each member has (near) equal importance, indicated by similar weights on the inputs. The equivalence classes can be identified by clustering antecedents with similar weights, and then by extracting rules in terms of the equivalence classes. In the final rule defining the behaviour of a neuron, members of an equivalence class can be interchanged without affecting the outcome.

Neither the feature-driven SVM approaches, nor the decompositional neural network approaches, can be directly used to extract rules from the voted perceptron model of Chapter 4. Although the perceptron is a single-layer neural network, the use of the DNF or k-DNF kernel implicitly introduces hidden layers whose connection weights are unknown; or, equivalently, it expands the input space into the higher-dimensional feature space of conjunctions of input features. The neural network methods rely on knowing the network structure and the connection weights, so would have to work with the expanded feature space and use the structure of a single-layer perceptron. In this scenario, the connection weight w_f on any feature $\mathbf{f}$ (a conjunction of input features) is just the prediction calculation made by the perceptron, so

$$w_f = \sum_{i=1}^{n} \alpha_i y_i K(\mathbf{f}, \mathbf{x}_i) \tag{6.1}$$

where K is either the DNF or k-DNF kernel, and the $\mathbf{x}_i$ are the support vectors. Unfortunately, working in the much larger feature space immediately brings (further) tractability problems, since the extraction methods operate on every feature: e.g., with the DNF kernel, the subset methods now run in time $O(2^{2^n})$ where n is the number of input features.

Since the kernel perceptron also produces support vectors and a decision function, the feature-driven SVM extraction methods could be applied to extract rules. One issue is that the perceptron’s support vectors are not necessarily located close to the decision boundary, as the SVM support vectors are. Methods designed for SVMs may therefore not work as well for perceptrons. More importantly, most of the feature-driven methods perform feature selection on features in the input space, and do not account for combinations of features being discriminative where individual features are not (Barakat and Bradley, 2007; Chen et al., 2007). As preconditions are conjunctions of features which jointly predict change but individually do not, such feature selection processes could ignore features which are actually relevant. Even in the case of Zhang et al. (2004, 2005a) where the rule extraction method was specifically designed for the DNF family of kernels, the possibility that weights are spread across all subconjunctions of a conjunction is not accounted for. The approach is still relatively successful, since it favours smaller conjunctions which can be combined to form larger conjunctions covering the true rules. However, this is likely to be why the method produces many rules.

6.1.3 Feature selection methods

Rule extraction is also closely related to the problem of feature selection, where the aim is to reduce the number of features used by a classifier during learning, in order to reduce training and prediction times, or to improve accuracy. Feature selection processes remove irrelevant or redundant features, a goal shared with rule extraction. In fact, since in their simplest form action preconditions are conjunctions of relevant features, removing the irrelevant features and conjoining the remainder is a reasonable strategy to generate rules. For feature selection to work as a form of rule extraction it is important that the features should be selected *according to the learning algorithm*, since it is the features useful to the learning algorithm which are required.

Feature selection methods are usually split into *filter methods*, *wrapper methods*, and *embedded methods* (Kohavi and John, 1997; Guyon and Elisseeff, 2003). Filter methods perform pre-processing to identify relevant features based on characteristics of the data, and so can be disregarded here, as the learning algorithm has no impact on the choice of features to discard.

In contrast, a wrapper method explores the full feature powerset lattice, using the learning algorithm as a black box to evaluate each subset of features it considers (and so is similar to the pedagogical methods above). In this context, the lattice exploration can use *forward selection*, where the search begins from the empty set, or *backward elimination*, where the search begins from a full set of features. The search algorithm chooses potential subsets of features, which are evaluated by estimating the accuracy of the learning method on the training set, when only those feature subsets are used as training data. Accuracy can be estimated, for example, by using cross-validation. Once an acceptable level of accuracy is achieved, the search terminates. Suitable simple search algorithms include hill-climbing or best-first search (Kohavi and John, 1997).

Provided evaluation is not too expensive, wrapper methods could also be adapted to extract rules from the perceptron models.

Embedded methods differ in that they are integrated into the learning algorithm itself, but many are also based on the wrapper approach of exploring the feature lattice. Of these, methods designed for SVMs are particularly closely related to the rule extraction problem, acting in a similar way to the decompositional methods previously discussed. Several approaches (e.g., Rakotomamonjy (2003); Weston et al. (2000)) are related to SVM Recursive Feature Elimination (SVM-RFE) (Guyon et al., 2002). SVM-RFE uses backward elimination and proceeds by first training the SVM, ranking each feature according to some ranking criterion and then deleting the feature with the lowest rank. The procedure is repeated until no features remain. For a linear SVM, the ranking criterion is w_i^2, where $\mathbf{w}$ is the weight vector calculated by the SVM. The criterion corresponds to the change in cost function resulting from removing the i-th feature. As noted by Rakotomamonjy (2003), this ranking criterion measures the sensitivity of $\mathbf{w}$ with respect to the inputs, similarly to methods used for neural network feature selection and rule extraction. For non-linear SVMs using a kernel, the ranking criterion is again the change in cost function when the i-th feature is removed. This would lead to computationally expensive retraining of the SVM, except that it is avoided by assuming that the set of support vectors remains the same. Instead only (parts of) the kernel matrix must be recalculated to account for the deleted feature.

6.2 Extracting preconditions from (k-)DNF kernel perceptrons

As discussed above, none of the existing decompositional rule extraction methods can be applied to the problem of extracting preconditions from the voted perceptron models learnt in Chapter 4. However, some of the underlying principles from the decompositional and feature selection methods can be used, namely, identifying discriminative features by using the weights assigned by the model, and using information provided by the support vectors and the decision function.

Working in the feature space of all conjunctions of fluents, each voted perceptron learns a weight vector, assigning a weight to each conjunction. It is useful to think of the feature space as a lattice where conjunctions or hypotheses are ordered by a generalisation relation. Depending on the representation used, the generalisation relation corresponds to different forms of the subsumption relations discussed in Chapter 2. When working in the vector representation, each hypothesis can be represented as a vector, in the same way as the states. Elements corresponding to fluents which are non-discriminative are set to N in the hypothesis. Thus, in the vector case, hypothesis h *subsumes* or *covers* an example e if whenever an entry in h has value 1 or -1, the entry at the same position in e has the same value. Here, the covering relation corresponds to propositional subsumption. Similarly, for the graph representation, each hypothesis can be represented as a graph where only nodes for those fluents (positive or negative) which are relevant to the hypothesis are present in the graph. Then a hypothesis h covers an example e if h is isomorphic to a subgraph of e. Here, the covering relation corresponds to OI-subsumption. In both cases, a set of hypotheses covers an example if every hypothesis in the set covers the example.

The approach discussed below extracts preconditions from a voted perceptron via decomposition approach, in that it requires access to the support vectors, and effectively determines discriminative features in each support vector by using the coefficients of the weight vector in the feature space. There are also pedagogical aspects to the approach, since it is a search through the lattice of possible hypotheses for a conjunctive precondition, where the model is used as a black box to decide which hypothesis to select next. However, the model must provide access to the full weight it calculates for an input, rather than just the final prediction, as it would in a purely pedagogical approach. Since the models predict changes to each fluent separately, after extracting preconditions for change to individual fluents, an additional step is required to combine the preconditions and effects into a full description of the action.

6.2.1 Rule extraction from individual perceptrons

The perceptron's model is entirely described by its support vectors. Each support vector is classed by the perceptron as positive or negative.[2] The positive support vectors are each instances of some rule the perceptron has learnt, and so they are used to "seed" the search for rules. It is assumed that each positive support vector is covered by exactly one conjunctive rule (but the full set of positive support vectors may be covered either by a single conjunction or a disjunction of conjunctions). The extraction process aims to identify and remove all irrelevant entries in each support vector, effectively

[2]Note the perceptron's classification may differ from the target values associated with the support vectors. For the purposes of seeding the rule search, the perceptron's classifications are used, but for the purposes of calculating the kernel function values, the original target values are used.

navigating through the lattice of hypotheses from the support vector towards the maximally specific covering rule. Then the set of rules are combined to form either a single conjunction or k-DNF, depending on the expected form of the preconditions.

Rules are extracted from a perceptron with a kernel K and a set of support vectors $SV = SV^+ \cup SV^-$, where SV^+ (SV^-) is the set of support vectors whose *predicted* values are 1 (-1). For any vector $\mathbf{x}$ in the subsumption lattice, its weight is defined to be the value calculated by the perceptron's prediction calculation before thresholding (Equation (6.1)). The predicted value for $\mathbf{x}$ is 1 if $weight(\mathbf{x}) > 0$ and -1 otherwise. A *child* of vector $\mathbf{x}$ is any distinct vector obtained by replacing a single element of x with the value N. Similarly, a *parent* of $\mathbf{x}$ is any vector obtained by replacing an N-valued element with either the value 1 or -1.

The basic intuition behind the rule extraction process is, as in both feature selection and the neural network subset methods, that more discriminative features will contribute more to the weight of an example. By repeatedly deleting the feature which contributes least to the weight, we should be left with the most discriminative features underlying an example, which can be used to form a precondition. An example of the process of extracting rules is shown in Figure 6.2, and an outline of the algorithm in Figure 6.1, as follows. Take each positive support vector v in turn, and aim to find a conjunction $rule_v$ which covers v and does not cover any negative training examples, but where every child of $rule_v$ covers at least one negative example. Construct $rule_v$ by a greedy algorithm which first takes v as a candidate rule and then repeatedly selects a new candidate rule by taking the child of the current candidate whose parents have the least difference in weight - that is, finding

$$\operatorname*{argmin}_{x_i \in \{x_1,\ldots,x_i,\ldots,x_n\}} (weight(x_1,\ldots,x_i,\ldots,x_n) - weight(x_1,\ldots,\neg x_i,\ldots,x_n)).$$

Removing the resulting x_i removes the least discriminative entry in the current candidate rule. At each step the new candidate rule is tested against the training examples. If it classifies a negative training example as positive, then the rule is too general and $rule_v$ is set to the previous candidate rule, otherwise the process repeats.

```
for $v \in SV^+$ do
    $child := v$
    while every training example which $child$ covers is positive do
        $parent := child$
        flip each valued element in $parent$ to its negation in turn
        pick the result $child$ whose parents have the least difference in weight
    $rule_v = parent$
```

Figure 6.1: Rule extraction algorithm

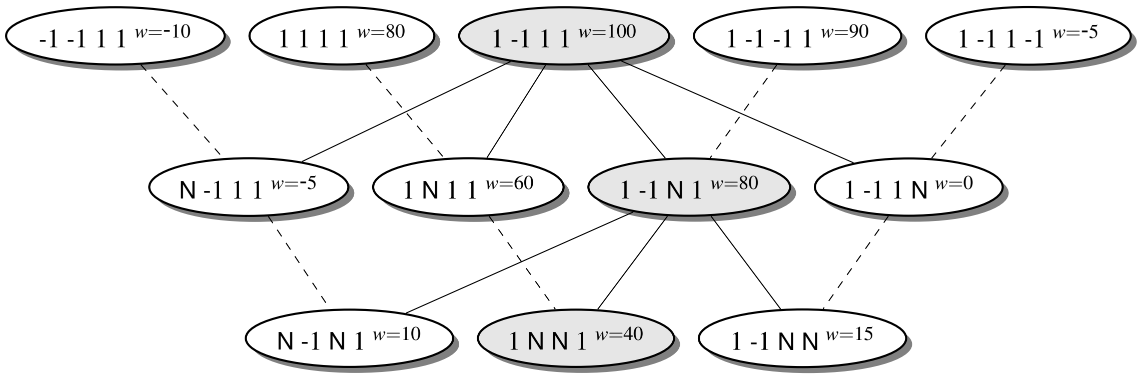


Figure 6.2: Part of the lattice of rule hypotheses, showing a simple example of the rule extraction process. Each node contains a vector corresponding to a possible precondition, and the weight w assigned to the vector by the voted perceptron model. Each level of the lattice contains vectors with one fewer feature than the level above. Lines join parent and children nodes: solid lines link the candidate parent rule at one level with its children in the level below, and dashed lines link children to their alternative parent. Shaded nodes are the preconditions selected at each iteration through the lattice. The positive support vector "seed" is the vector $\langle 1\ \text{-}1\ 1\ 1 \rangle$ with weight 100. The child whose parents have the least weight difference, the vector $\langle 1\ \text{-}1\ \text{N}\ 1 \rangle$, is chosen as the next candidate rule. The process ends with the rule $\langle 1\ \text{N}\ \text{N}\ 1 \rangle$ as both children have a negative counterexample in the training data (not shown).

The result is a set of rules for each action, predicting when a particular entry in the output vector changes. There may be many rules, up to one per positive support vector, each consisting of a set of preconditions which, if satisfied, predict the entry in the output vector will change. The number of rules can be reduced by identifying and deleting duplicates, and also via merging of rules when combining rules across different entries of the vector (see Section 6.3).

An example of the set of rules extracted for the BlocksWorld `stack` action is shown in Table 6.1. For reference, the true rule is shown in Figure 6.3. Since the per-effect rules are extracted from a perceptron trained on 1000 examples from a world with 5% noise and 25% partial observability, it is expected there will be some errors in the rules. Indeed, there is an extra effect (`on-table ?x1`), while many of the individual rules have additional incorrect preconditions. However, all of the preconditions and effects in the true rule are present. Overall, for this particular training instance, the perceptron model produces an F-score of 0.72 on the test set, while the extracted per-effect rules perform at a similar level, with an F-score of 0.71.

Per-effect rule	Weight

`arm-empty changes when:`
`(not (arm-empty)) (not (on-table ?x1))`
	8

`clear ?x1 changes when:`
`(not (clear ?x1)) (holding ?x1) (not (on ?x1 ?x2))`
`(not (clear ?x1)) (not (on-table ?x1)) (not (on ?x2 ?x1))`
`(clear ?x1) (on-table ?x1)` `(not (on ?x1 ?x2)) (clear ?x2) (not (on ?x2 ?x1))`
	14
	12
	8

`on-table ?x1` `changes when:`
`(not (on-table ?x1)) (not (on ?x1 ?x2))`
`(not (arm-empty))` ***`(on-table ?x1) (not (clear ?x2))`***
	6
	4

`holding ?x1 changes when:`
`(holding ?x1)`
	15

`on ?x1 ?x2 changes when:`
`(not (on ?x1 ?x2))`
	3

`clear ?x2 changes when:`
`(clear ?x2)` ***`(on-table ?x2)`*** `(not (holding ?x2))`
`(not (arm-empty)) (not (clear ?x1)) (holding ?x1) (not (on ?x1 ?x2)) (clear ?x2)`
`(not (clear ?x1)) (clear ?x2)` ***`(not (on-table ?x2))`***
	12
	6
	2

Table 6.1: Per-effect rules generated for the BlocksWorld `stack` action from 1000 examples in a world with 5% noise and 25% observability. Fluents in bold are neither in, nor implied by, the true action specification.

True `stack` specification

```
(:action stack
  :parameters (?x1 ?x2)
  :precondition (and (holding ?x1) (clear ?x2))
  :effect (and (arm-empty) (clear ?x1) (not (holding ?x1)) (on ?x1 ?x2)
               (not (clear ?x2))))
```

Figure 6.3: Specification of the true BlocksWorld `stack` action, for comparison.

Note that the search through the lattice has to move from specific to general hypotheses rather than from general to specific. A general-to-specific search could operate by starting with the empty hypothesis and navigating towards a specific support vector by flipping vector elements from N to the corresponding value in the support vector, choosing the highest weighted parent each time. However, the general-to-specific option only identifies the highest weighted conjunction which could be a rule, and it is not necessarily the correct rule or the only rule. In contrast, the specific-to-general direction defines a set of possible rules, at least one of which will be the "correct" rule.

The process is similar to performing a subset method on the single layer neural network represented by the perceptron model, where the inputs are in the expanded feature space. Such a network has all of the inputs connected to a single node, which outputs the final classification. Only one input is ever active for any state observation: the input which exactly matches the true and false fluents in the observation, with the remainder unobserved. The weight on a connection with input $\mathbf{x}$ is $weight(\mathbf{x})$. A crucial difference is that the perceptron-based method has support vectors available to it, and so can use these to focus the search only on weight combinations which cover the positive support vectors. Also, the process does not identify all possible subsets which are guaranteed to give a positive weight, but instead just finds the superset of those subsets. Both of these substantially reduce the number of weight combinations which have to be considered.

6.2.1.1 Computational complexity

In fact, if the state vectors have n elements, and r elements are removed, then discovering each precondition requires $\sum_{i=0}^{r}(n-i)$ flips of an entry in the vector. Each time an entry is flipped, the weight of the resulting hypothesis is calculated: the weight calculation takes $O(n)$ time for the DNF kernel and $O(nk)$ for the k-DNF kernel (Sadohara, 2002). The time complexity associated with flips is therefore $O(n^3)$ or $O(n^3k)$. Every time an entry is removed, the resulting hypothesis is tested against all t training examples. If there are s support vectors this takes $O(tsn)$ or $O(tsnk)$ time for the DNF and k-DNF kernels respectively. The number of support vectors is equal to the number of mistakes the perceptron made during training and so is known to be polynomial in t for the k-DNF kernel, but not for the DNF kernel. So overall, the rule extraction process is polynomial in the number of training examples and the length of the state vectors when using the k-DNF kernel, but not the DNF kernel.

6.2.2 Incorporating voting

So far the rule extraction process has only used the final perceptron produced by the learning models, rather than the full voted perceptron. A number of ways to extend the process to the full voted perceptron can be envisaged. I take the simplest approach of replacing the weight calculation with the voted perceptron prediction function. Whereas in the search guided by the standard perceptron, the aim is to navigate through the space of hypotheses by removing entries which contribute least to the weight of a support vector, in the voted perceptron analogue, the aim is to remove entries which contribute least to the vote determining the prediction for the support vector, i.e.,

$$weight(\mathbf{x}) = \sum_{i=1}^{n} c_i \, sign \sum_{j=1}^{i} y_j \alpha_j K(\mathbf{x}_j, \mathbf{x}).$$

Another possibility would be to perform the rule extraction process for each of the perceptrons in the full voted perceptron model, and then combine the rules, taking into account the number of votes associated with each perceptron. However, it is not clear how best to combine the large number of weighted rules produced in this way.

6.3 Combining rules across output elements

Having extracted a precondition (a disjunction of conjunctions) for each output element of a vector and action, it remains to combine these into rules for each action. Recall that an action rule has an action name with parameters, a set of preconditions and a set of effects. The preconditions and effects are conjunctions of fluents whose parameters are defined in terms of the action parameters. There may also be conditional effects, where if additional preconditions are satisfied, additional effects occur. In the following section, we consider the case where it is known that there is a single conjunctive rule for each action. It is left to the discussion (Section 6.5.2) to consider how this preliminary step could be extended to disjunctive preconditions, which may be modelled by allowing a single action to have multiple conjunctive rules.

6.3.1 Building pure conjunctive rules

In noiseless domains (partially or fully observable), if it is known that there is a single conjunctive rule for each action, the rules for individual outputs can trivially be combined by respectively concatenating all of the preconditions and all of the effects into a single precondition and effect. Fluents in the resulting precondition cannot contradict each other, provided there is only one rule per action and no noise in the domain, as contradictory examples are never seen by the model, and hypotheses which do not cover any examples are never considered. The resulting preconditions will tend to be over-specific, but the alternative of taking the intersection of all preconditions will fail in partially observable domains, as the preconditions will often be empty. Consequently, a back-tracking step is used to try to identify unnecessary preconditions. For any combination of rules, this will involve generating less specific alternatives, suggested by the rules, and testing whether these alternatives are acceptable, according to some scoring function.

If there is noise, but still only one rule per action, rule combination is further complicated by the possibility of conflicts in the preconditions or effects. The rule combination step must incorporate some means of resolving such conflicts. This may produce several potential rule combinations, which again must be tested by the scoring function to determine which is best, and whether it is better than not combining the rules at all. Even once an acceptable rule combination is identified, the rule may still contain incorrect preconditions or effects introduced by noise. As before, backtracking combined with the scoring function is used to reject rules where this is likely.

6.3.1.1 The rule combination algorithm

The rule combination process aims to build a single conjunctive rule for each action. It operates on all rules ($\langle precondition, effect \rangle$ pairs) for an action produced by the earlier rule extraction process. For each action it first initialises the baseline rule to a default rule consisting of the precondition of the highest weighted rule, and no effects. The baseline rule is then refined by attempting to combine it with each of the remaining pairs in turn, in order of highest weight. Each time the process must generate a suit-

```
$allRules :=$ rules from rule extraction process
$hwr := HighestWeighted(allRules)$
$baseline := (hwr.pre, \varnothing)$
$locks = \varnothing$
while $allRules \neq \varnothing$ do
   $next := HighestWeighted(allRules)$
   $allRules := allRules \setminus \{next\}$
   if $CompatibleEffects(baseline, next)$ then
      if $CombinePrecons(baseline, next, newpre, locks)$ then
         $BacktrackPrecons(baseline, next, newpre)$
         if $AcceptPrecons(baseline, next, newpre)$ then
            $baseline.pre := newpre$
         if $AcceptEffect(baseline, next.eff)$ then
            $baseline.eff := baseline.eff \cup next.eff$
         $BacktrackEffects(baseline)$
```

Figure 6.4: Rule combination algorithm: pure conjunctions

able candidate rule, and then decide whether to accept or reject the candidate as the new baseline. An outline of the rule combination algorithm is given in Figure 6.4, with supporting functions in Figures 6.5 and 6.6. The functions are described in the following section.

6.3.1.2 Assessing compatibility

Before attempting to combine the current baseline rule with a new rule, checks are made to ensure this is worthwhile (*CompatibleEffects*). If the new rule's effect contradicts any effect in the baseline rule,[3] the new rule is automatically rejected, since at this point we assume only one rule per action, and the higher weighted baseline rule is more likely to be correct.

6.3.1.3 Generating new rules

Rules can be combined on both effects and preconditions, and each of these is considered separately. First, an attempt is made to combine the rule preconditions with the

[3]Since effects in the learnt model are changes, it may seem strange that a change to an element *e* in the effect vector given by the new rule could conflict with a change to the same element in the effect of the baseline rule. However, since the rules are being converted into PDDL, where effects are absolute, the value of an element in the effect vector is determined by the corresponding element in the preconditions. Therefore if the corresponding element in the baseline precondition contradicts the same element in the new rule's precondition, the effects conflict.

baseline preconditions, to form a set of candidate preconditions (*CombinePrecons*). If the preconditions of the two rules do not contradict, they can be combined by conjunction to form the candidate preconditions. Otherwise, the non-conflicting fluents are conjoined as before, and an attempt is made to reconcile the conflicts.

For each conflicting fluent, three variants of the candidate preconditions are created, where the fluent has value N, 1 or -1. The weight $weight_{eff}$ (for each effect *eff* in the baseline effects) of each variant is calculated. The preferred variant is the one where the fluent has value N, which means the fluent is omitted from the preconditions, since this value indicates a non-discriminative feature. However, a variant is only acceptable if the weight of the resulting rule is positive for all of its effects. If accepted, the fluent is also locked at that value, to prevent later, possibly noisy rules, from resetting the value without any checks on whether N is still an acceptable value. This relies on the assumption that when the weight calculation indicates an entry is irrelevant for one rule vector, it remains irrelevant for another. If the N-variant is unacceptable, then the (1)-valued or (-1)-valued cases are considered, provided they have positive weights on all the effects. If both variants are acceptable, whichever variant has the highest average weight over all the effects is selected. If neither variant is acceptable then the conflict is unresolved for this fluent. As long as the conflicts on every fluent are resolved, the rule combination process can continue with the new candidate rule. If not, the whole attempt at combining the baseline with the current rule is abandoned, since no suitable combination can be formed.

At this stage, a backtracking step (*BacktrackPrecons*) generates a number of alternative, less specific preconditions from the candidate preconditions. For each fluent in the candidate which does not exist in the original preconditions, an alternative candidate precondition is constructed, without that fluent. If the scoring function (*AcceptPrecons*) rates the baseline precondition as worse than the alternative precondition, then the fluent is removed from the candidate preconditions.

6.3.1.4 Testing new candidate rules

Next, the resulting candidate rule is compared with the baseline rule, using a separate scoring function for the preconditions (*AcceptPrecons*) and effects (*AcceptEffects*). If the precondition scoring function rates the candidate preconditions as acceptable, then they become the new baseline preconditions. The process is similar for the candidate effects. Candidate preconditions may be accepted without the effects, and vice versa.

The precondition scoring function (*AcceptPrecons*) accepts the candidate precon-

```
function $CombinePrecons(baseline, next, precons, locks)$
   $conflicts := \varnothing$
   for all $p \in baseline.pre$ do
      if $p \in baseline.pre \wedge \neg p \in next.pre$ then
         $conflicts := conflicts \cup p$
      else
         $precons := precons \cup p$
   if $\forall e \in baseline.eff, weight_e(precons) > 0$ then
      $locks := locks \cup conflicts$
   else
      for all $p \in conflicts$ do
         $posweight := \sum_{e \in baseline.eff} weight_e(precons \cup \{p\})$
         $negweight := \sum_{e \in baseline.eff} weight_e(precons \cup \{\neg p\})$
         if $\forall e \in baseline.eff, weight_e(precons \cup \{p\}) > 0$ then
            if $posweight \geq negweight$ then
               $precons := precons \cup \{p\}$
         else if $\forall e \in baseline.eff, weight_e(precons \cup \{\neg p\}) > 0$ then
            if $negweight > posweight$ then
               $precons := precons \cup \{\neg p\}$
         else
            return $false$
   return $true$


procedure $BacktrackPrecons(baseline, next, precons)$
   $backtracks := \varnothing$
   for all $p \in precons \setminus \{baseline.pre\}$ do
      $alternative := precons \setminus \{p\}$
      if $AcceptPrecons(baseline, next, alternative)$ then
         $backtracks := backtracks \cup \{p\}$
   $precons := precons \setminus backtracks$


function $AcceptPrecons(baseline, next, precons)$
   for all $e \in baseline.eff \cup next.eff$ do
      if $weight_e(precons) \leq 0$ then
         return $false$
      else if $covers_e(precons) = 0$ then
         return $false$
      else if $F_{precons,e} < 0.95 \times F_{baseline.pre,e}$ then
         return $false$
   return $true$
```

Figure 6.5: Supporting functions used in the rule combination algorithm (preconditions).

ditions if they are no worse than the existing preconditions, when used to predict any of the effects in the candidate rule. It makes use of both the weight calculation and the training examples. Both coverage and weight should be considered, as weight alone may permit rules for which there is no evidence in the training data (a positively weighted rule can potentially cover no training examples), while coverage alone may allow negatively weighted rules.

Firstly, a combination of two rules can be rejected if, for any of the resulting effects e, the resulting precondition p does not have a positive weight: $weight_e(p) \leq 0$. Similarly, a rule combination can be rejected if, for any effect, it does not cover any training example, as then there is no evidence in the training data to support the rule: $covers_e(p) = 0$. Here the definition of coverage is relaxed to account for incomplete rules and examples. A rule r consists of an action a, preconditions $r.pre$ and effects $r.eff$, while a training example x consists of an action a, a prior state $x.state$ and changes $x.changes$. Rule r covers example x at effect e ($covers_e(r,x)$) if none of the fluents in the example state contradict the fluents in the rule preconditions, and e is in both the example state changes and the rule effects. Formally,

$$covers_e(r,x) \iff compatible(r.pre, x.state) \wedge e \in r.eff \cup x.changes,$$
$$\text{where } compatible(A,B) \iff (f \in A \Rightarrow \neg f \notin B).$$

Now the coverage of rule r on the training set is defined to be:

$$covers_e(r) = |\{x : covers_e(r,x)\}|.$$

Secondly, the precondition scoring function uses differences in precision and recall to identify and reject any rule which performs significantly worse than its comparison rule. It rejects rules where either the precision or recall drops substantially for any of the elements in the rules' effect vectors. Since precision and recall is a trade-off, the comparison is made using the F-score for precondition pre at effect eff: $F_{pre,eff}$. If the new F-score $F_{new,eff}$ is less than ε_p times the baseline F-score $F_{baseline,eff}$, on any effect eff, then the new rule is rejected. In the results presented here, ε_p is set to 0.95.

The effects scoring function (*AcceptEffect*) is similar to the preconditions scoring function, in that it compares F-scores. Instead of comparing the F-scores of two different rules on each effect, it takes a single rule and compares the F-score of one selected effect against the F-score for every other effect. This identifies effects which are inconsistent with the other effects in terms of precision and recall. In particular, effects which occur in far fewer examples than other effects are identified in this way: these

```
function $CompatibleEffects(baseline, (pre, e))$
  if $e \subseteq baseline.eff$ then
    if $(e \in baseline.pre \wedge \neg e \in pre) \vee (\neg e \in baseline.pre \wedge e \in pre)$ then
      return $false$
  return $true$

procedure $BacktrackEffects(baseline)$
  $backtracks := \varnothing$
  for all $e \in baseline.eff$ do
    if $\neg AcceptEffect(baseline, e)$ then
      $backtracks := backtracks \cup \{e\}$
  $baseline.eff := baseline.eff \setminus backtracks$

function $AcceptEffect(baseline, testeff)$
  for all $e \in baseline.eff$ do
    if $F_{baseline.pre,testeff} < 0.5 \times F_{baseline.pre,e}$ then
      return $false$
  return $true$
```

Figure 6.6: Supporting functions used in the rule combination algorithm (effects).

are likely to be caused by noise, or could be conditional effects. An effect is rejected by the function if its F-score is less than ε_e times the F-score on any other effect of the same rule. In the results presented here, ε_e is set to 0.5.

Finally, the effects backtracking function (*BacktrackEffects*) tests if any of the effects should be removed from the rule, in light of the new preconditions. Each effect is tested by the effects scoring function and, if rejected, removed from the rule's effects.

6.3.1.5 Computational complexity

The computational complexity of rule extraction depends on the number of initial rules, and the number of preconditions and effects in the baseline and candidate rules, as well as on the complexity of the weight and coverage calculations. As mentioned in section 6.2.1.1, the number of initial rules for a single effect is bounded by the number of support vectors, which is polynomial in the number of training examples t. The number of preconditions and effects is bounded by the length of the state vector n (and in practice is substantially lower). The weight calculation, when using the k-DNF kernel, is as previously noted, $O(nk)$ for each effect, while the covers calculation operates on each training example, and compares every element in the rule vector to

Function	Complexity
CombinePrecons	$O(\#precons).O(\#effects).O(weight) = O(n^3k)$
AcceptPrecons	$O(\#effects).(O(weight)+O(covers)) = O(n^2(t+k))$
BacktrackPrecons	$O(\#precons).O(AcceptPrecons) = O(n^3(t+k))$
CompatibleEffects	$O(\#effects) = O(n)$
AcceptEffect	$O(\#effects).O(covers) = O(n^2t)$
BacktrackEffects	$O(\#effects).O(AcceptEffect) = O(n^3t)$

Table 6.2: Complexity of each of the supporting functions, where the length of the state vector is n, the number of training examples is t, and the perceptron model uses the k-DNF kernel. $O(weight)$ and $O(covers)$ denote the computational complexity of the weight and covers calculations.

every element in the training example's state vector, and so takes time $O(nt)$. The time complexities of the various supporting functions are summarised in Table 6.2. *BacktrackPrecons* dominates, and is polynomial in n, t and k. Since the outer loop runs for each rule, and the number of rules is polynomial in the number of training examples and the number of effects, the full rule extraction process is also polynomial in n, t and k.

6.3.1.6 Converting to PDDL

At this point in the process, the preconditions and effects are still written as vectors rather than as readable rules, e.g., in PDDL format. The conversion from a vector precondition to a PDDL precondition is straightforward: given the action and its parameters, the fluent and parameters corresponding to a particular position in the vector are known, and the value at that position gives the value of the fluent. In the case of the effects, the values only indicate whether a particular fluent changes, but not what the change is from or to. If the fluent also exists in the preconditions then this can be used to determine the correct value of the fluent in the effects; otherwise, the value can be obtained from the support vector from which the rule for the element of the effect vector originated.

The scenario of an effect occurring without a corresponding precondition is relatively rare, typically occurring when the domain is fully observable and noiseless. In these cases, the rule extraction process is often able to discard fluents from the precondition, if they are implied by other fluents in the precondition. It is for this reason that taking precondition values from an originating support vector is successful, since the support vector will have a fully specified, correct, state description. The information the support vector provides is effectively a proxy for background information about the

Actions/Fluents	Precondition vector	Changes vector	Action/Preconditions	Effects
pickup(?x1)				
putdown(?x1)				
stack(?x1 ?x2)	1		stack(?x1 ?x2)	
unstack(?x1 ?x2)				
(arm-empty)	-1	1	(not(arm-empty))	(arm-empty)
(holding ?x1)	1	1	(holding ?x1)	(not(holding ?x1))
(on-table ?x1)				
(clear ?x1)		1	(not(clear ?x1))[SV]	(clear ?x1)[SV]
(on ?x1 ?x2)		1		[unknown]
(on ?x1 ?x3)				
...				
(holding ?x2)				
(on-table ?x2)				
(clear ?x2)	1	1	(clear ?x2)	(not(clear ?x2))
(on ?x2 ?x2)				
(on ?x2 ?x3)				
...				

Figure 6.7: Conversion of precondition and changes vectors to PDDL format for a hypothetical BlocksWorld `stack` example. The Actions/Fluents column shows the actions and fluents corresponding to each position in the precondition and changes vectors. Entries in the vectors with value 0 are omitted for clarity. Entries in the preconditions vector can be mapped directly to fluents, since -1 indicates a negated fluent, and 1 a positive fluent. Entries in the changes vector indicate only that the fluent changed, but not what its new value is, and so can only be directly mapped if the value is also given in the preconditions vector. Otherwise it may be possible to identify the value from an originating support vector, which may also lead to an additional precondition, as in the case of `(clear ?x1)`. It may not always be possible to identify the effect value, in which case the effect is not added to the PDDL description (see `(on ?x1 ?x2)`).

domain; equally then, if background information were available it could also be used to fill in the missing data. For similar reasons, once any effects have been added, the preconditions can be augmented with negations of any fluents occurring in the effects list.

An example of the PDDL conversion is shown in Figure 6.7. Suppose the rule combination algorithm produces a rule, consisting of the precondition vector and changes vector shown in Figure 6.7. The PDDL preconditions can be read directly from the preconditions vector:

```
((not arm-empty) (holding ?x1) (clear ?x2)).
```

Any effect with a corresponding entry in the preconditions has a value which is simply the negation of the value in the preconditions, giving an initial effects vector:

```
(arm-empty (not (holding ?x1)) (clear ?x2)).
```

The values of the remaining effects cannot be derived from the preconditions. Instead, we look to an example which the rule was derived from, a support vector which seeded the original search for rules. As a highly-weighted example is likely to be more reliable, the highest weighted of these support vectors is selected to provide the effect values. Supposing this highest weighted support vector in our example is `((not (arm-empty)) (not (clear ?x1)) (holding ?x1))`, the additional effect `(clear ?x1)` can be added to the effects. The final effect relating to the value of `(ON ?x1 ?x2)` is lost, however, as there is no information about which value it should take.

6.4 Experiments

Rule extraction was performed on the 3-DNF kernel perceptron models learnt in Chapter 4. In all experiments, the rules are assumed to have purely conjunctive preconditions. The quality of the extracted rules is evaluated, firstly, as in previous experiments, by considering the F-score of the rule predictions on test sets of 2000 examples, and secondly, by directly comparing the extracted rules with the true domain rules, using a measure of error rate (Zhuo et al., 2010).

The error rate for a single action is defined as follows. The number of errors on the preconditions of the action E_{pre} is the sum of the number of extra preconditions and the number of missing preconditions. The number of errors on the effects E_{eff} is defined similarly. In practice E_{pre} is obtained by comparing the rule preconditions to a minimal set of true preconditions (to determine whether any necessary fluents are missing) and also to a maximal set of true preconditions (to determine any extra fluents). The maximal set includes any fluent implied by the minimal true preconditions. E_{eff} is obtained likewise. The total number of possible fluents (in the preconditions or effects), T, is just the number of possible grounded fluents in the domain, whose arguments are any of the action parameters, and nothing else. Then the error rate of a single action a is defined to be:

$$Error(a) = \frac{1}{2}\left(\frac{E_{pre} + E_{eff}}{T}\right)$$

and the error rate of a domain model with a set of actions A is defined to be:

$$Error(A) = \frac{1}{|A|}\sum_{a \in A} Error(a).$$

While the F-score of the rule predictions on the test set gives an indication of how the rules may perform in practice, the error rate gives an absolute measure of the difference between the generated and actual rules.

6.4.1 Results

In general, the F-scores for predictions made by the rules closely parallel the F-scores of the perceptron models for both partially observable, noiseless domains (see Figure 4.7) and partially observable, noisy domains (see Figures 4.8 and 4.9). In fact, there is no statistically significant difference between the predictions made by the perceptron models and those made by the extracted rules.[4]

The error rates similarly indicate that the learnt models are close to the actual STRIPS domain definitions, falling below 0.1 after around 10,000 examples in all cases. In particular, the correct STRIPS model is produced by rules in less than 2,000 training examples when the domains are fully observable and noiseless.

There are two key aspects in which the results differ from the raw perceptron models. Firstly, in the Rover domain, the rule predictions are worse than the perceptron predictions. This result is expected, given that some actions in the Rover domain were identified as requiring conditional rules in a representation which codes effects as changes rather than absolute values (Section 3.3). Since the rules used here are not conditional, they are not expressive enough to fully represent the Rover domain. For example, in each of the ten training cases, after 20,000 examples the `take_image` action is learnt in a fully observable, noiseless world as either:

```
(:action take_image
   :parameters (?x1 ?x2 ?x3 ?x4 ?x5 )
   :precondition (and (at ?x1 ?x2) (visible_from ?x3 ?x2) (calibrated ?x4 ?x1)
                      (supports ?x4 ?x5) (on_board ?x4 ?x1))
   :effect (not (calibrated ?x4 ?x1)) )
```

(7 cases, where 3 have slightly more specific preconditions than the true rule), or

```
(:action take_image
   :parameters (?x1 ?x2 ?x3 ?x4 ?x5 )
   :precondition (and (at ?x1 ?x2) (visible_from ?x3 ?x2) (calibrated ?x4 ?x1)
                      (supports ?x4 ?x5) (on_board ?x4 ?x1)
                      (not(have_image ?x1 ?x3 ?x5)))
   :effect (and (have_image ?x1 ?x3 ?x5) (not (calibrated ?x4 ?x1)) ))
```

(3 cases, where 2 have slightly more specific preconditions than the true rule). Around 70% of the errors on the test domains after training on 20,000 examples are due to the `TAKE_IMAGE` action, indicating that the lack of support for conditional effects is affecting the performance of the rules.

[4]Repeated measures ANOVA, $p > 0.05$.

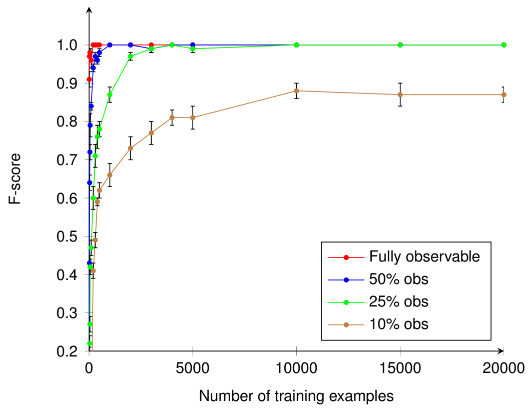
Secondly, across all the domains, rules learnt at the 5% noise level under full observability produce worse predictions than rules learnt with the same level of noise where observations were incomplete. This result is slightly counter-intuitive, since rules learnt from complete observations are usually better than those learnt from incomplete observations, and rules learnt in noiseless environments are usually better than those learnt in noisy environments. It might be expected that the effect of combining noise and incomplete observations would be to worsen the predictions further (which is the case for the raw perceptron models).

The discrepancy relates to how rules are combined from the individual voted perceptrons. Each voted perceptron learns a precondition for the change of a single effect. Without noise or partial observability, each voted perceptron for the same action will learn the same preconditions. With noise or partial observability, different preconditions may be learnt. For example, consider the precondition of the BlocksWorld `(pickup ?x1)` action, where `arm-empty`, `(clear ?x1)` and `(on-table ?x1)` are the discriminative features, while `arm-empty`, `(clear ?x1)`, `(on-table ?x1)` and `(holding ?x1)` are the features which change. Suppose the highest weighted rules produced by the voted perceptrons are:

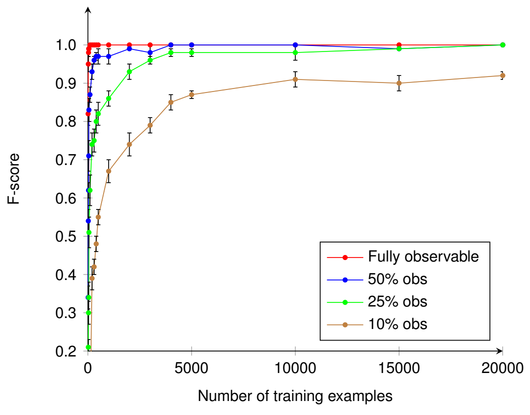
`arm-empty` $\wedge$ `(clear ?x1)` predicts change to `arm-empty`, and,

`arm-empty` $\wedge$ `(on-table ?x1)` predicts change to `(on-table ?x1)`.

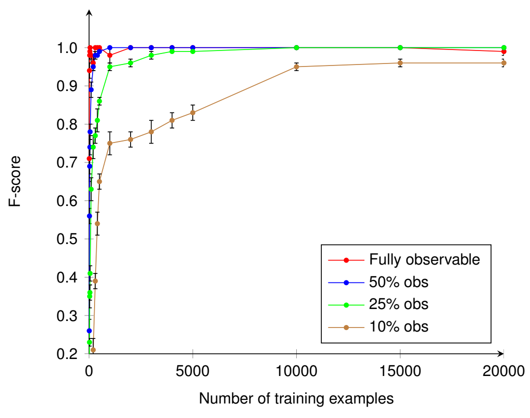
During rule combination, the first of these could be taken as an initial estimate of the `pickup` rule. It will produce many errors on the training set, in the form of false positives, as it ignores the values of `(on-table ?x1)` when making predictions. If rule combination now attempts to refine the rule, it will consider the precondition `arm-empty` $\wedge$ `(clear ?x1)` $\wedge$ `(on-table ?x1)` and the changes `arm-empty` and `(on-table ?x1)`. Without noise, the new rule will have fewer false positives and no additional false negatives in the predictions, so its F-score will be higher than the F-score of the original rule, and the new rule will be accepted. If the training set is noisy, however, some positive examples correctly predicted by `arm-empty` $\wedge$ `(clear ?x1)` will be false negatives for the refined rule, as they will not match the rule on `(on-table ?x1)`. Conversely, some of the previous false positives will now be true negatives. The number of new false negatives will be the number of training examples where the initial state was `arm-empty` $\wedge$ `(clear ?x1)` $\wedge$ `not(on-table ?x1)`, and `arm-empty` changed, while the number of new true negatives will be the number of training examples where the initial state was the same but `arm-empty` did not change. It is possible that the increase in true positives is outstripped by the increase in false negatives, and then the F-score for the refined rule is lowered and it may not be accepted. The longer the true rule, the worse this effect is. If the training set is also partially observable, to a greater extent than it is noisy, then it is more likely that the entries at `(on-table ?x1)` will be unobserved than noisy. Then many of the previous false negatives will be true positives, as an unobserved value is assumed always to match a rule. The F-score for the refined rule is therefore higher than in the noisy, but fully observable case, and so the rule is more likely to be accepted.



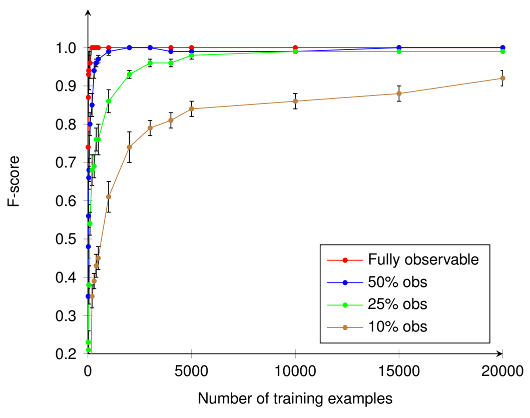
(a) BlocksWorld



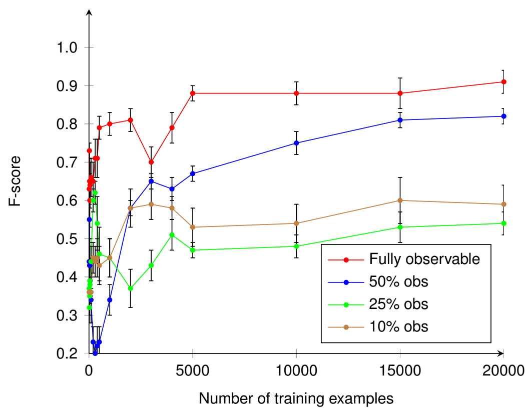
(b) ZenoTravel



(c) Depots

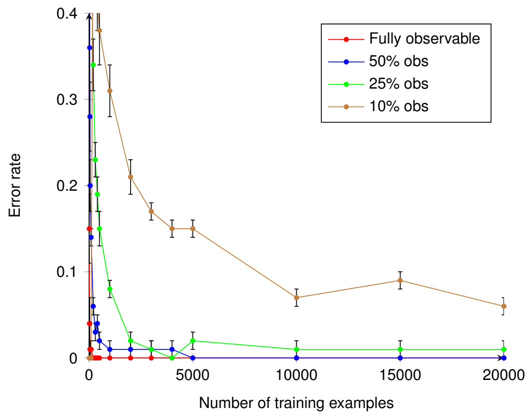


(d) DriverLog

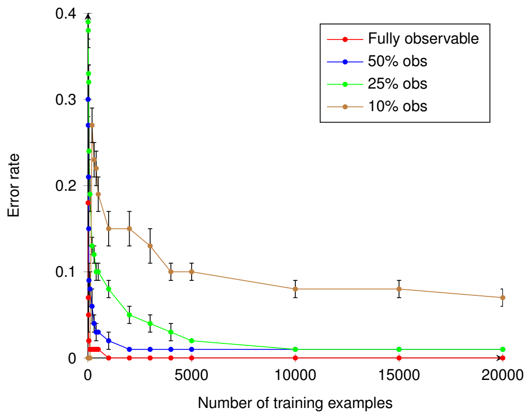


(e) Rover

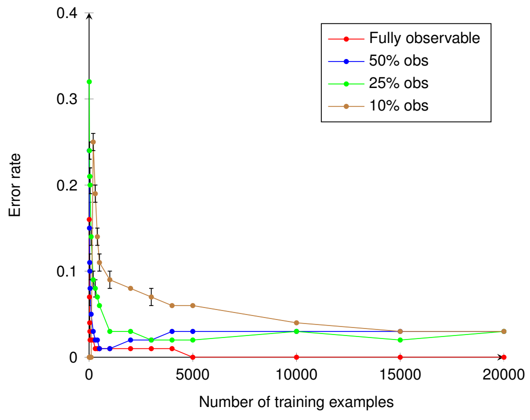
Figure 6.8: Results from learning rules in partially observable, noiseless, simulated planning domains, using a voted perceptron with the 3-DNF kernel. Classifiers were trained on varying numbers of examples, and rules were extracted and then tested on 2,000 fully observed, noiseless examples from worlds in the same domain as the training examples, but with more objects.



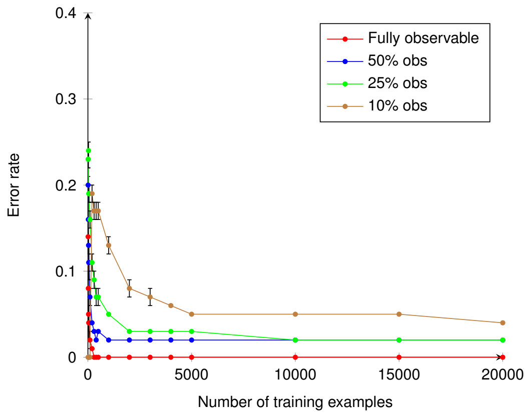
(a) BlocksWorld



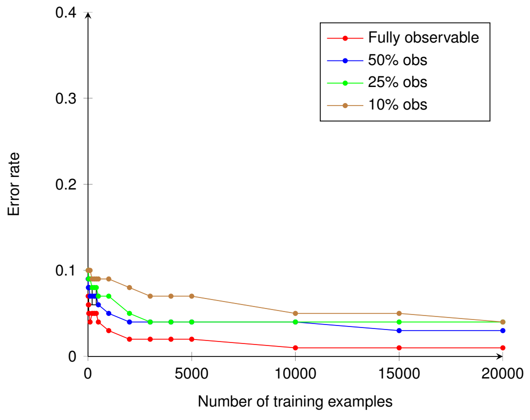
(b) ZenoTravel



(c) Depots

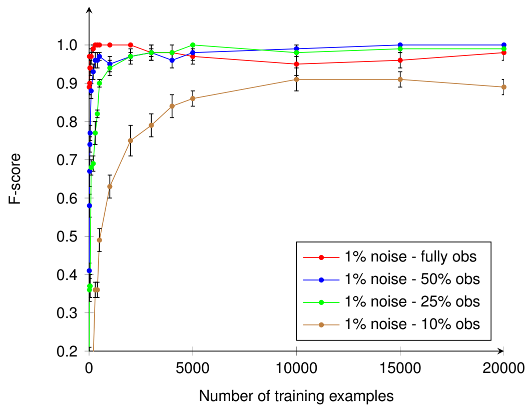


(d) Driverlog

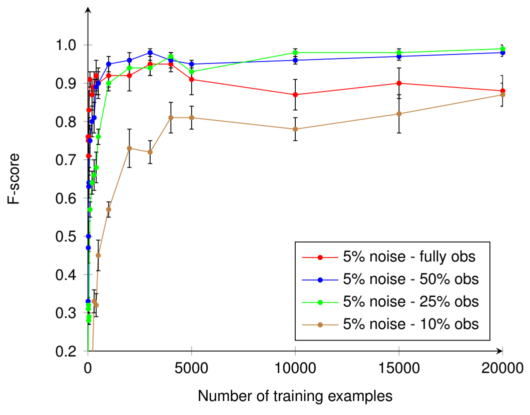


(e) Rover

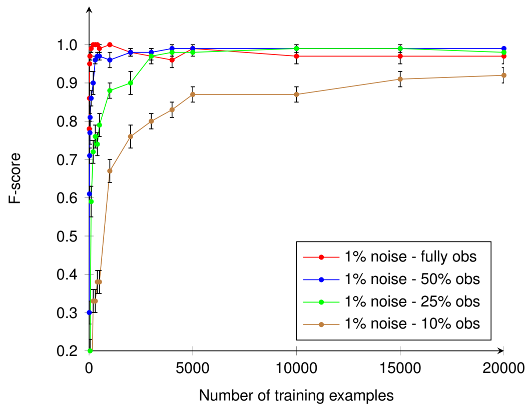
Figure 6.9: Error rates for different levels of observability in the domain.



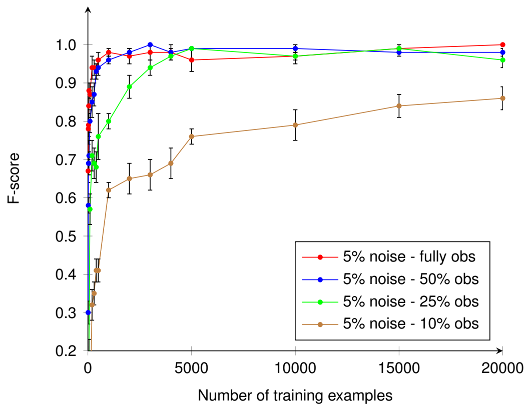
(a) Blocksworld (1% noise)



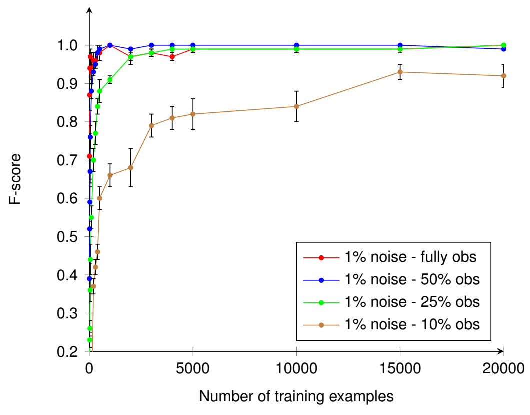
(b) Blocksworld (5% noise)



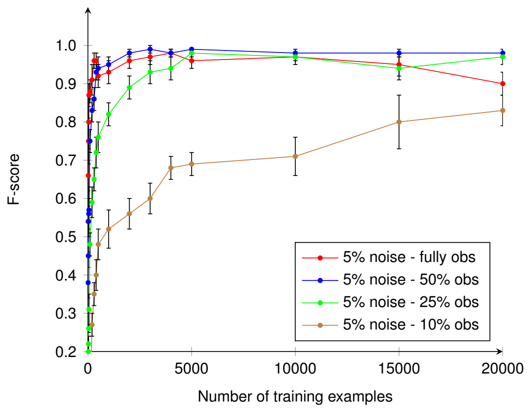
(c) ZenoTravel (1% noise)



(d) ZenoTravel (5% noise)

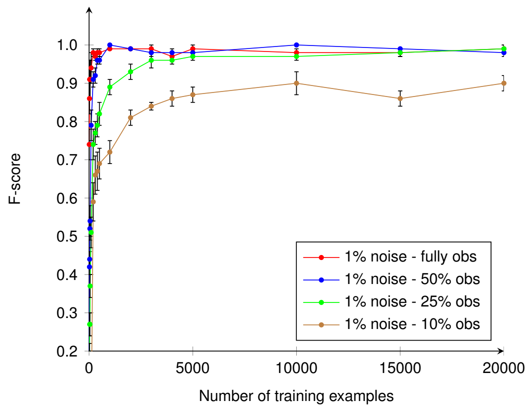


(e) Depots (1% noise)

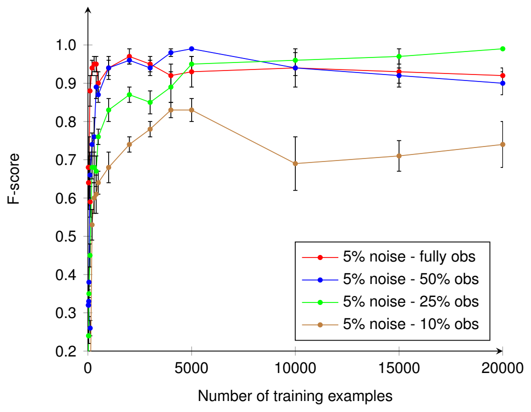


(f) Depots (5% noise)

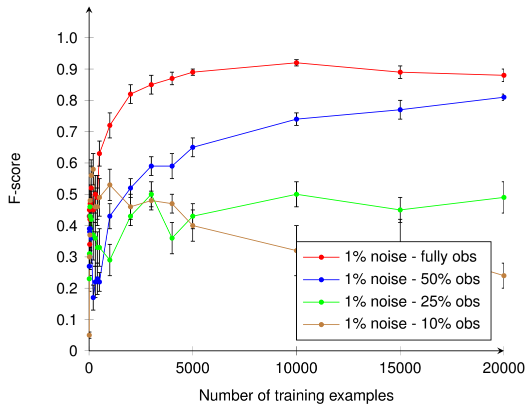
Figure 6.10: Results from learning rules in simulated planning domains with varying levels of noise (1%, 5%) and observability (100%, 50%, 25%, 10%), using a voted perceptron with the 3-DNF kernel, and assuming pure conjunctive rules.



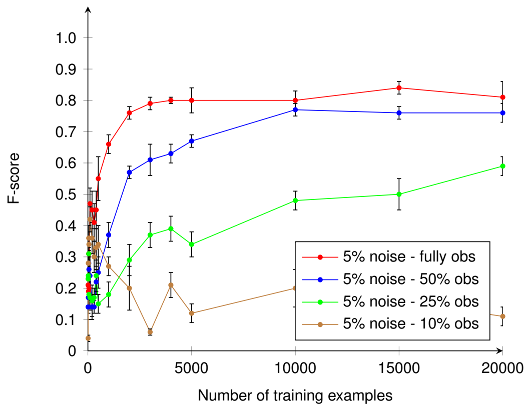
(a) Driverlog (1% noise)



(b) Driverlog (5% noise)

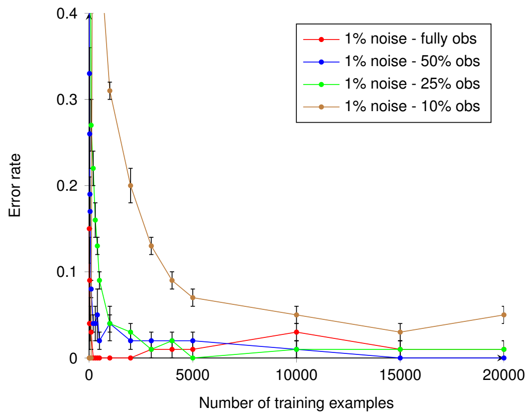


(c) Rover (1% noise)

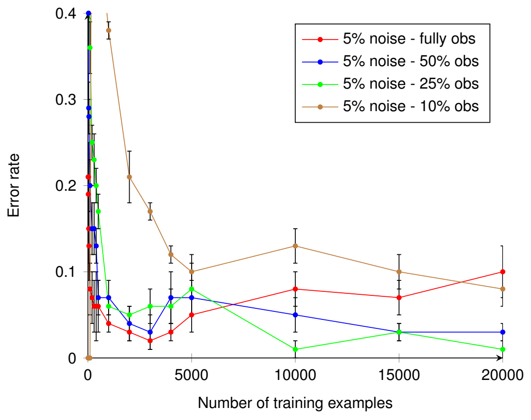


(d) Rover (5% noise)

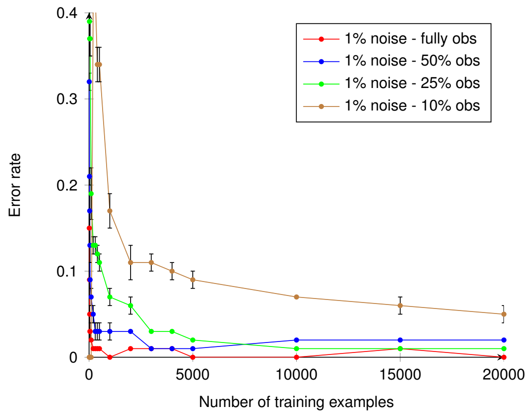
Figure 6.11: Results from learning rules in simulated planning domains with varying levels of noise (1%, 5%) and observability (100%, 50%, 25%, 10%), using a voted perceptron with the 3-DNF kernel, and assuming pure conjunctive rules.



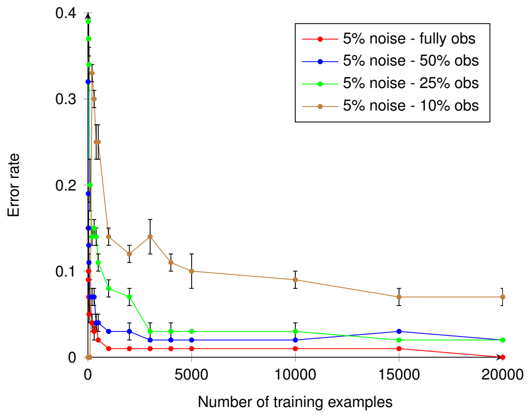
(a) Blocksworld (1% noise)



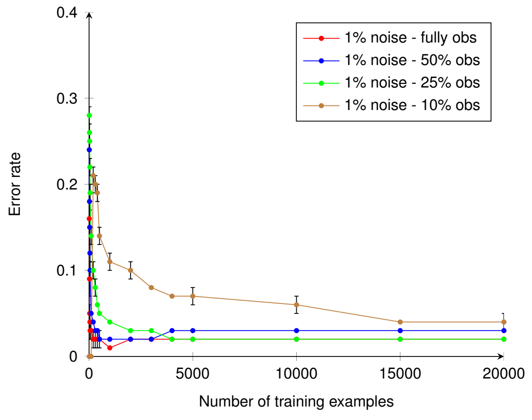
(b) Blocksworld (5% noise)



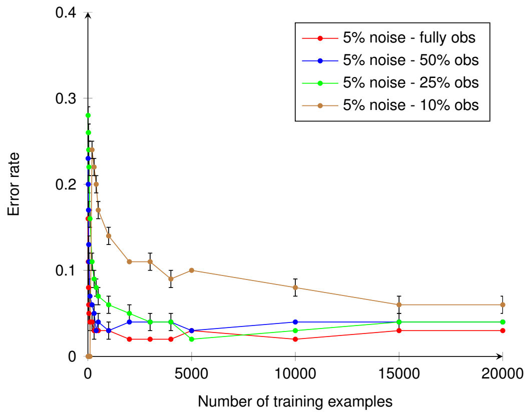
(c) ZenoTravel (1% noise)



(d) ZenoTravel (5% noise)

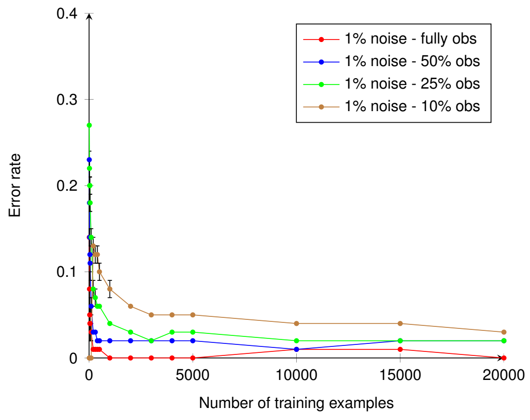


(e) Depots (1% noise)

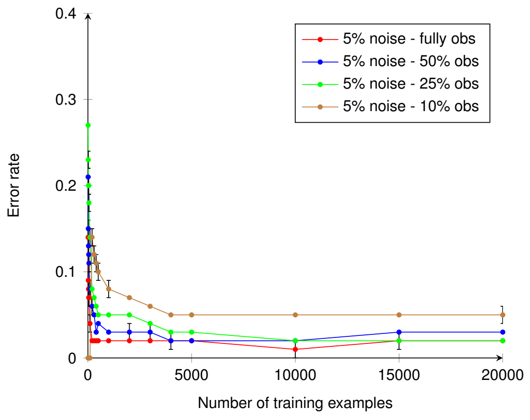


(f) Depots (5% noise)

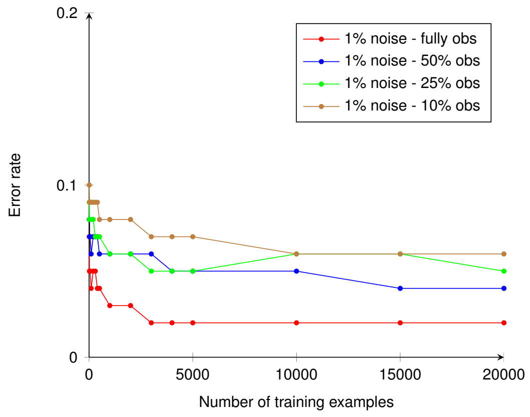
Figure 6.12: Results from learning rules in simulated planning domains with varying levels of noise (1%, 5%) and observability (100%, 50%, 25%, 10%), using a voted perceptron with the 3-DNF kernel, and assuming pure conjunctive rules.



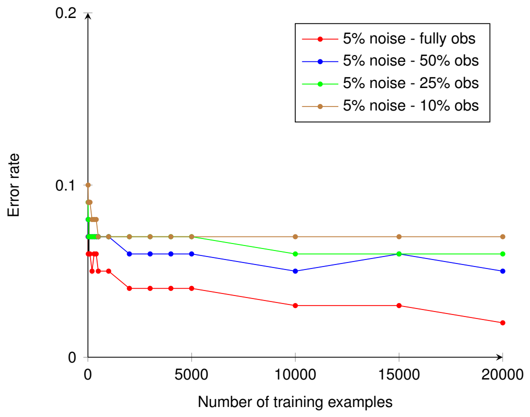
(a) Driverlog (1% noise)



(b) Driverlog (5% noise)



(c) Rover (1% noise)



(d) Rover (5% noise)

Figure 6.13: Results from learning rules in simulated planning domains with varying levels of noise (1%, 5%) and observability (100%, 50%, 25%, 10%), using a voted perceptron with the 3-DNF kernel, and assuming pure conjunctive rules.

6.5 Discussion

The experiments demonstrate that the extracted rules perform as well as the raw perceptron models across a variety of different levels of noise and partial observability. The rules can therefore be substituted for the perceptron models without loss of performance, and will make faster predictions, since usually determining whether a rule covers an example will be faster than carrying out the full voted perceptron calculation.

The actual rules are often approximate, in that they may have missing or additional preconditions or effects. In terms of the impact such inaccuracies might have, the F-scores show that for randomly generated examples, those for which the rules predict wrongly, occur rarely. Similarly, the error rates indicate that the errors are often caused by a single erroneous fluent in the full domain description.

Even with these inaccuracies, the domain models are useful. For instance, it has been shown that humans find it much easier to modify approximate rules to the correct domain model, than to generate the model from scratch (Zhuo et al., 2010). A set of rules describing domain dynamics in PDDL form can be processed by most planners, paving the way for combined autonomous learning and planning in noisy and incomplete domains, providing the planner itself can operate in such domains, and can handle approximate models. A planner using rules acquired in a noisy or partially observable domain is likely itself to be working with noisy or incomplete examples. In these situations the outcomes of actions even with the true domain model may be unpredictable, so it seems reasonable to expect that the planner also can work with approximate examples. Some planners with these capabilities exist: for instance, the PKS planner can handle incomplete knowledge (Petrick and Bacchus, 2002) while model-lite planning (Yoon and Kambhampati, 2007) works with non-deterministic domains and approximate models. Planning with the approximate models generated by the rule extraction process will probably work best where the learner and planner are working in tandem, as in model-lite planning, so that errors discovered in the course of planning can be used to update the model and the corresponding rules. Further work is required to determine how inaccuracies in the domain model will affect downstream users of the rules.

By compressing the information in the perceptron models, the rules enable efficient storage of what has been learnt, supporting communication of domain information between agents, and potentially between different functions of the same agent. This observation has further implications. An important area for future work is to consider how rules might be converted *back* into the weighted lattice. While rules provide compact storage, and enable communication of the learnt models, it should be possible to modify them at a later stage. For instance, a partial rule communicated from another agent will only aid learning if it can be combined with the experiential learning process. Similarly, previous experience in an environment, resulting in incomplete rules describing the world model, is more useful if the learning process can take advantage of the rules already learnt.

6.5.1 Relation to other work

In terms of rule extraction from voted perceptrons, or related classifiers such as SVMs and neural nets, the work presented in this chapter is closest to Zhang et al. (2004, 2005a). They also aim to extract classification rules from an SVM trained with the DNF kernel and some variants. Likewise, the approach makes use of the support vectors and the weight function to identify useful rules. However, positive and negative classification rules are extracted separately by decomposing the weight function into a positive weight function (the part of the sum generated by the positive support vectors) and a negative weight function (the part of the sum generated by the negative support vectors). For the positive weight function, it initially constructs a set R_1 of all conjunctions of length 1 whose weights are greater than some user-defined threshold. It then incrementally constructs R_{n+1} from R_n and R_1 so that $r \in R_{n+1}$ if r is the conjunction of $r' \in R_n$ and $r'' \in R_1$ such that the weight of r is greater than the threshold. The final set of rules produced by the method is the union of the R_i. The same process is repeated for the negative weight function.

The approach has several issues affecting its suitability for constructing rules from planning domains. Firstly, it will tend to generate many rules, since in order to extract a rule of length n, it must extract all of its 2^{n-1} subconjunctions. Secondly, it is likely to generate contradictory rules, since the positive rules are generated without considering the negative weights and vice versa. Lastly, the weight value calculated for a conjunction is individual, that is, it does not include the weights of its subconjunctions. This is a critical issue, because, especially under partial observability, individual weights tend to be higher on shorter subconjunctions and lower on longer subconjunctions of a rule. As a result, the true rule conjunctions are likely to be rejected because their weights are too low.

The rule extraction process also has strong parallels to the SVM-RFE algorithm. Both operate using backward elimination, and determine which feature to remove next by ranking features according to the sensitivity of the weight function to each feature. Crucially, however, while SVM-RFE works with a set of features, rule extraction works with a set of features and their corresponding values, namely an example in the form of a positive support vector. Thus rule extraction seeks features which are important in determining the class of a *specific* example, while SVM-RFE seeks features which are important in determining the class of *any* example. The distinction is more important when there are disjunctive rules, since SVM-RFE will locate features which are discriminative in any rule, while rule extraction will locate features discriminative in the context of a particular example. Therefore rule extraction generates sets of features corresponding to different disjunctions of a rule, while SVM-RFE only generates a single set of features.

The error rates calculated for the extracted rules permit a subjective comparison with the ARMS algorithm of Yang et al. (2007), as they also calculate error rates for their experiments. The ARMS algorithm learns STRIPS models, but only uses positive examples in the context of successful plans, and only observes positive fluents. The model of partial observability is similar to the one used here, as after each action a small number of fluents are observed. For partial observability at level $p\%$, the number of observed fluents is $p\%$ of the estimated number of fluents in the full state, where the estimate is the size of the set of all fluents ever seen in the preconditions, add and delete lists of the full plan. ARMS was tested on the Depots, Driverlog, ZenoTravel, Rovers and Satellite domains. Notably, the error rates for ARMS are much higher than for rules extracted from the voted perceptron models, suggesting that if there is a choice between learning from known plans with ARMS and learning from random exploration with voted perceptrons, the latter option may be preferable. Rovers proves particularly difficult, never achieving an error rate below 0.6 for partial observability between 10% and 90%, with approximately 3700 training examples. The Depots, DriverLog and ZenoTravel domains fare better, but still have error rates at higher levels than the rules extracted from voted perceptrons. With around 1600, 4000 and 4000 training examples respectively, error rates are above 0.2 at 10% observability, and remain above 0 at 90% observability.

6.5.2 Building disjunctive preconditions and conditional effects

The rule combination process laid out in this chapter is limited to pure conjunctive preconditions. While such preconditions are adequate for STRIPS rules, the perceptrons are able to learn more expressive disjunctive rules and conditional effects which a more sophisticated rule combination process could exploit. This would allow rules for more complex domains to be generated.

Additionally, since disjunctive preconditions support the existence of multiple rules for the same action label, the requirement that actions are distinguished by their labels could be relaxed. So, for example, the rules will support action X with preconditions a, b and c as well as action X with preconditions c and d, with the specific actions differentiated by their preconditions instead of their labels. Then the unrealistic assumption that the action labels completely partition the action space can be dropped. In the limit, actions could be represented as a set of component features, such as motor primitives, enabling the learning process to leverage similarities between actions, in the same way as it already leverages similarities between preconditions, in order to make predictions.

6.5.3 Deriving probabilistic effects

Modelling probabilistic effects requires the algorithm to consider multiple possible effects of any action, however the perceptron model converges on the most prevalent effect for each action, while less prevalent effects are essentially ignored (assumed to be noise). Thus until now the learnt rules have been of the form:

$$a(\bar{x}) : pre(\bar{x}) \rightarrow \mathit{eff}(\bar{x}),$$

where action a is applied to some set of terms $\bar{x}$. If the preconditions $pre(\bar{x})$ hold then the action results in changes described by $\mathit{eff}(\bar{x})$.

If in reality the domain is stochastic, it would be desirable to also identify alternative, less probable outcomes, namely to produce a set of *noisy deictic rules* (Pasula et al., 2007) for each action a. Each rule is of the form:

$$a(\bar{x}) : pre(\bar{x}) \rightarrow \begin{cases} p_0 : \mathit{eff}_1(\bar{x}) \\ \quad\vdots \\ p_{n-1} : \mathit{eff}_n(\bar{x}) \\ p_n : \text{noise outcome} \end{cases}$$

where the action has $n+1$ outcomes with corresponding probabilities such that $\sum_{i=0}^{n} p_i = 1$. An important aspect of noisy deictic rules is the noise outcome, which covers rare and noisy events which may be difficult or undesirable to model. Here it is anticipated that noise outcomes may be populated by outcomes resulting from low reliability classifiers.

A first step to generate a rule with probabilistic effects would be to first run the existing learning and rule extraction processes. This will lead to a single explicit rule for the action. It gives a precondition $pre(\bar{x})$ and the most probable outcome $\mathit{eff}_1(\bar{x})$, as a conjunction of effects. The problem is then to identify lower probability outcomes of the action in states which satisfy $pre(\bar{x})$. Additionally, once a set of outcomes has been identified, a probability must be assigned to each one.

In terms of identifying different possible outcomes, a simple approach would be to use the first, most probable rule to eliminate matching training examples, followed by retraining on the reduced training set, and extracting the next most probable rule, constraining the preconditions to match $pre(\bar{x})$. The process could be repeated until no further outcomes are found. However, this approach is quite inefficient, since it requires multiple training runs on the training data.

A more efficient approach would be to use the existing classifiers without retraining, as follows. Recall that each classifier is associated with a set of support vectors. While the classifiers only generate one prediction (i.e. one most likely outcome), the support vectors *do* contain data about less frequent outcomes. During learning by the voted perceptron, the hypothesis is continually adjusted by support vectors which either are or are not covered by the most probable rule. This rule prevails in rule extraction simply because on average the adjustments keep the hypothesis near to it. Removing the adjustments corresponding to the most probable rule from every classifier, by removing the support vectors covered by the rule, allows the remaining support vectors and weights to define a classifier for the next most probable rule.

Identifying support vectors covered by a rule is straightforward for complete, noiseless training examples, but is more difficult with the introduction of partial observability or noise, where a support vector may only partially match a rule, and may even partially contradict it. However, the rule combination process provides an alternative means of identifying support vectors to eliminate. Since at each iteration rule combination either incorporates a support vector into the current rule, or rejects the support vector, the set of support vectors considered to be covered by a rule are just those support vectors which were incorporated into the rule during rule combination.

It remains to assign specific probabilities to the set of possible outcomes of a rule. Within the subset of training examples covered by the same precondition, if the outcomes always cover disjoint training examples then the proportion of examples covered by each outcome may be used as an estimate of the respective probabilities. Otherwise Pasula et al. (2007) describe, in their *LearnParameters* algorithm, a gradient ascent method which can be used if training examples are covered by multiple outcomes: it calculates the probability distribution which maximises the log likelihood of the training examples covered by the rule. Additional measures would be needed to distinguish between noise and low probability outcomes, which may be achieved through the reliability score mechanism discussed in Section 5.1.3.

Noisy deictic rules can be used directly for planning. They have been used, for example, in an RMDP framework, where planning is performed by reasoning in a grounded relational domain (Lang and Toussaint, 2010). Furthermore, Lang and Toussaint (2010) discuss how noisy deictic rules may be converted to probabilistic PDDL rules, thus also enabling their use in other probabilistic planners such as RFF (Teichteil-Königsbuch et al., 2010).

6.5.4 Extending to the graphical representation

The rule extraction process presented above is specific to the vector representation of world states. Extending this approach to the graphical representation would enable the model to learn more complex preconditions consisting of existential conjunctive or k-DNF concepts. It is to be expected that moving to the graphical representation will present tractability problems, since similarity comparisons in the graphical representation will require subgraph isomorphism calculations.

There are some positive results in PAC-learning existential conjunctive and k-DNF concepts in structural domains with Boolean relations. Haussler (1989) gives a PAC learning algorithm for existential conjunctive concepts, assuming a fixed bound on the number of objects in a state, no noise, and the ability to make *subset queries*. A subset query is when the PAC learner formulates a hypothesis about what the target concept is and asks an oracle whether the hypothesis is contained in the target concept. In domains where the preconditions are constrained to be conjunctions, the model learnt in Chapters 4 and 5 can act as an oracle to answer subset queries (so Haussler's algorithm operates as a pedagogical rule extraction algorithm). Here the target concept is the precondition learnt by the model, rather than the real world precondition. Haussler's algorithm can therefore be applied to generate the preconditions for each individual effect. A post-processing step would then be needed to construct full PDDL rules from the separate precondition-effect pairs for each action.

Additionally, existentially quantified k-DNF concepts are PAC-learnable, where the number of variables in the expressions is bounded (Valiant, 1985), although the complexity is exponential in k. While 3-DNF rules could be extracted for individual effects, it would fall on the post-processing rule combination step to combine these into fewer rules with more conjunctions. The existence of these positive PAC-learning results suggests that it may be possible to tractably extract rules from models using the graphical representation.

6.6 Summary

The rule extraction approach outlined in this chapter shows that it is possible to extract STRIPS-like rules from voted perceptrons learning in noisy and partially observable STRIPS domains. The method follows the principles of several existing methods for extracting either rules or discriminative features from neural networks and SVMs, in that it recursively identifies discriminative features in the support vectors, using the weights assigned by each perceptron model. The derived per-effect rules are combined using a heuristic, based on the F-score of candidate rules on the training data, to determine which features should be combined to produce a final action rule.

The resulting rules are high quality, and close to the actual STRIPS rules for the domains. The rules are therefore suitable for use by planners which use PDDL domain descriptions, making it possible to combine the learning and planning processes. This in turn would support autonomous exploration and learning of an environment, by enabling a planner to direct exploration, rather than the random exploration currently employed by the learner.

There are several potential directions for further development of the rule extraction mechanism. The approach has the potential to produce more expressive rules, by extending it to extract disjunctive rules and conditional effects, and also rules with probabilistic effects. Furthermore, the process could be reversed: rules learnt from previous experience, or acquired from other agents, could be integrated with the learning process, either directly into the perceptron learning model, via the lattice weights, or incorporated into the rule combination step. The ability to use rules acquired from other sources is critical for an autonomous agent to participate in collaborative learning. The same mechanism may also be useful for transfer learning.

Chapter 7

Conclusions and Future Work

This thesis considers how an autonomous agent can learn grounded action representations in real world domains. The approach taken here depends on a central claim that noisy, incomplete observations can be handled by using a two-stage learning process, where first an implicit model is learnt and then explicit rules are extracted from the model. This claim is supported by the work presented in Chapters 4, 5 and 6. Chapters 4 and 5 showed how implicit action models of STRIPS and extended STRIPS domains could be learnt from noisy, incomplete observations, while Chapter 6 showed how explicit rules could be extracted from the learnt models.

A subsidiary claim is that action models can be learnt using standard classifiers, by encoding states in a representation based on deictic reference. In the classical STRIPS case in Chapter 4 this leads to an attribute-value representation of the problem and action model learning via voted kernel perceptrons using the k-DNF kernel. In the extended STRIPS case in Chapter 5 this leads to a graphical representation and a novel graph kernel based on deictic references, which together support action model learning, again via voted kernel perceptrons. The resulting algorithms can learn in classical STRIPS domains and extended STRIPS domains which include negative preconditions, conditional effects and universally quantified effects.

7.1 Contributions

The main contributions of this thesis are:

A new approach to learning action models: The two-stage approach to learning action models provides a means to handle noisy, incomplete observations while also generating comprehensible rules describing the learnt action model. The split learning means that a fast, incremental learning algorithm can initially be used to learn an implicit action model, while enabling a slower, possibly batch algorithm to learn explicit rules. Furthermore, the work of inducing rules is much simpler because the rules are not learnt directly from noisy, incomplete observations, but from observations which are generated by an implicit learnt model, and which are not subject to noise or partial observability. As an incremental learning approach which can operate in noisy, partially observable domains, this represents a practical algorithm for autonomous agents learning the dynamics of their world.

A novel graph kernel: The k-sitgraph kernel defined in Chapter 5 is a generalisation of the k-DNF kernel, based on deictic references. The generalisation allows the kernel to operate on graphical representations of states. The experiments in Chapter 5 demonstrate that this kernel supports learning of accurate action models. In terms of computational efficiency, the kernel is competitive with state-of-the-art graph kernels when applied to situation graphs.

A novel rule extraction method: The rule extraction algorithm presented in Chapter 6 differs from other rule extraction methods in that it uses the voted perceptron support vectors to seed the search through the hypothesis space of possible preconditions. At the individual classifier level, the principle of this approach may be applicable to kernels other than the k-DNF kernel. Once individual rules are extracted from each classifier, the algorithm uses them to seed a further search through the hypothesis space, to produce a final rule. When tested on STRIPS domains, the algorithm produces rules appropriate for use in a standard planner. Although only tested on voted perceptron models learnt using the k-DNF kernel, there is scope to generalise the algorithm for use on voted perceptron models learnt using situation graphs.

Evaluation against IPC planning domains: Both the learning algorithms and rule extraction method were evaluated against a set of benchmark planning domains taken from the International Planning Competition.

7.2 Future work

Some of the potential future directions have been discussed in earlier chapters, but these have related directly to the aspects of the learning algorithm discussed in the individual chapters, rather than across the learning as a whole. In this section I briefly recap the earlier discussions, before considering future work which impacts the whole approach described in this thesis.

7.2.1 Extensions to the learning algorithm

In Chapter 4 I proposed that the initial STRIPS learning algorithm should be extended to more complex domains, and that explicit rules should be extracted from the implicit models. These proposals were implemented in Chapters 5 and 6. The main issue remaining in Chapter 5 is the limitation of the representation to first-order deictic references, which it was suggested could be overcome by introducing predicate invention or allowing effects to recursively reference actions.

In the rule extraction process, disjunctive preconditions and conditional effects are not extracted although they are available in the implicit models. This must be addressed in order to apply the full learning approach to more realistic domains. Additionally, in Chapter 6 I discuss the possibility of reversing the rule extraction process to allow rules provided by external sources to be integrated into the learning process. Further extensions discussed below apply to the learning algorithm as a whole.

7.2.1.1 Directed Exploration

In this thesis I have used a basic random exploration strategy, where the next action is chosen uniformly at random. In general this is inefficient and subject to the law of diminishing returns. Furthermore, in some domains where “interesting” actions are only present in a small part of the state space, random exploration may fail to generate any useful examples at all. In the experiments, the effects of random exploration can be seen in the plateauing of F-scores after several thousand training examples (e.g. Section 4.4.1).

To overcome some of these limitations, one possibility is to adapt an approach from the reinforcement learning community. The problem of choosing which action to execute next is commonly found in reinforcement learning, where an agent must choose between selecting an action which will maximise its reward based on its current knowledge (exploiting), versus selecting an action which will increase the agent’s knowledge of the world and potentially increase future rewards (exploration): the *exploration-exploitation tradeoff*. Recent work has begun to extend solutions to the exploration-exploitation problem in MDPs and POMDPs to the relational case.

In MDPs, the E^3 (Kearns and Singh, 1998, 2002) and R-MAX (Brafman and Tennenholtz, 2002) algorithms give near-optimal solutions to the exploitation-exploration problem. E^3 tracks *known states*, defined to be states which have been visited more often than some threshold. Whenever the system is in an unknown state, E^3 executes *balanced wandering*, where it takes the action which has been taken least from the current state. In known states, E^3 will exploit, if it can find a policy which stays within the known states with high probability, or alternatively, plan an exploration in an alternative MDP where unknown states have high values, thus pushing exploration into unknown states. R-MAX takes a similar approach, tracking known and unknown states, and assigning the highest reward (R_{max}) to unknown states. Unlike E^3, it does not make an explicit choice between exploration and exploitation.

These algorithms have also been extended to handle factored MDPs (Guestrin et al., 2002; Kearns and Koller, 1999), while recent work has produced solutions for the POMDP case (Doshi-Velez et al., 2012; Cai et al., 2009). However, these representations are non-relational, and as before (Section 2.4), are not appropriate for the relational case because of the lack of generalisation across objects, and because of the difficulties of working in the large state spaces characteristic of relational worlds.

As noted by Walsh (2010) and Lang et al. (2010), there has been very little research on the exploration-exploitation problem in the context of relational domains. Such research has focussed on the fully observable case. The REX algorithm (Lang et al., 2010) is based on E^3, with modifications to handle relational worlds. Whereas a state was considered known in E^3 if it had had a certain number of visits, in REX a density function is estimated from previously seen states and a state is considered known if the function assigns it a probability above some threshold. Similarly, the choice of action to perform during balanced wandering is based not on the number of executions, but on a confidence measure derived from the number of experiences covered by the (current) abstract rule for the action. In contrast to the heuristic approach of REX, Walsh (2010) derives sample complexity bounds on techniques based on extensions to the R-MAX algorithm. Both approaches handle stochastic actions but assume a fully observable world.

Finally, Rodrigues et al. (2011) carry out ε-*active* exploration. For a proportion ε of the trials, a random action is chosen. For the remainder, they perform *active exploration* where an action is selected which the agent believes will lead to a change in the current action model, by increasing the set of counter-examples used to generate rules. The choice of action is relatively conservative: an action may be selected if all of its effects are false in the current state, which intuitively may lead to a generalisation of the original action rule. This approach additionally assumes deterministic actions as well as a fully observable world.

The exploration-exploitation problem is thus an area of active research which has only recently begun to be explored, and the available techniques are designed for world observations which are complete. As a consequence, most work in relational action models learns, as in this thesis, from traces of actions and observations obtained via random walks through the state space. Recently, however, Lang et al. (2010); Lang (2011) used the rule-learning algorithm described by Pasula et al. (2007) in tandem with REX to learn action models on a variety of simulated and real-world robot tasks. Learning was faster in comparison to ϵ-greedy exploration where the agent exploited with probability $\epsilon(= 0.1)$ and randomly explored otherwise.

It would also be possible to integrate the rule-learning approach described in this thesis with REX or ϵ-greedy exploration in a similar fashion. In fact, the incremental nature of my approach makes it better suited to this type of exploration (the approach of Pasula et al. (2007) is batch). An alternative direction, closer to the work of Rodrigues et al. (2011), would be to work with the rule lattice of Chapter 6, which encodes information about visited states in the form of weights assigned by the support vectors to individual elements of the lattice. Working from the position of an existing action rule in the lattice, it may be possible to identify parts of the lattice with little evidence for including or excluding them from coverage by the rule. From there, states which would provide more evidence can be identified, thereby generating more useful support vectors for the learning process.

7.2.1.2 Noise models

The blocking process model used to simulate noise and partial observability (Section 3.5) is a relatively poor approximation of perceptual uncertainty, since it assumes all fluents are equally likely to be affected by noise or to be obscured. This ignores dependencies between fluents in terms of both correlations and world constraints. In reality some fluents are likely to be correlated, for instance, in a domain with balls and blocks on a table, balls are more likely to be on the table than on a block, because a ball is more likely to fall on the table than to balance on a block. Similarly, some fluents will be constrained to depend on others, such as in BlocksWorld where whenever `(armempty)` is true, `(holding x)` should be false. Furthermore, it is likely that particular objects may be affected by sensing difficulties (which also affects fluents involving those objects), and also that particular relations may be more difficult to determine than others. For instance, in the real robot domain in Chapter 5, the fluent `ontable` is much less prone to errors than `isin` because of the difficulties in detecting objects when they are within other objects.

One alternative would be to adopt the perceptual strategy of taking repeated observations, often used in robot localisation. For example, position probability grids (Burgard et al., 1996) are populated by repeated measurements from multiple sensors to estimate state to some required degree of certainty. State here is typically location but could be generalised to the symbolic state space or the underlying sensor space. Such a strategy (or assumption of) could significantly lower the level of noise on the observed fluents, improving the performance of the action learning model described in this thesis.

However, there are issues to overcome at both the symbolic and sensor levels. Working with state estimations in this way at the symbolic, relational level requires techniques to handle the explosion in the size of the state space, such as already discussed in the context of learning transition functions (Section 2.4.4) and exploration (Section 7.2.1.1) in relational POMDPs. Working at the sensor level avoids the dimensionality problem but requires a perceptual function to generate the symbolic representation: as previously discussed (Section 3.5), a full solution to generating perceptual functions for relational data is still an open research topic. A perceptual function learnt via supervised learning could be used, but if an oracle is available to label states then this could as easily be used to label the states for the action learning process itself.

Again at the level of continuous observations from sensors, a more sophisticated approach would be to begin with a Gaussian noise model on the raw observations. By propagating the effects of this model through the perceptual function, noise models for the symbolic percepts could be derived. However, the success of this approach depends on the nature and existence of the perceptual function.

7.2.1.3 Introducing a world model

The learning process discussed in this thesis could be improved by the introduction of world knowledge, independent of operator preconditions and effects. For example, in some worlds there will be predicates which can be derived from other, more basic, predicates. Derived predicates do not need to be predicted, but can be derived from the predictions of the basic predicates, trivially improving the efficiency of learning. Conversely, the addition of derived predicates to observations can improve prediction quality, as they effectively merge fluents which otherwise would have to be learnt as separate fluents in the precondition. Furthermore, a world model which describes known dependencies between fluents could be used to identify certain combinations of fluents as illegal. In turn, this would reduce the number of potential conjunctions to be considered during learning, (and, in particular, the number of subgraphs in the case of the graphical representation) thereby making learning more efficient.

7.2.1.4 Parallelism

The structure of the learning algorithm in Chapters 4 and 5 is *embarrassingly parallel*: learning for each individual classifier could be performed in parallel without requiring inter-process communication. Similarly, the initial stages of rule extraction in Chapter 6 could be carried out concurrently. With the rapidly developing potential for parallel processing in computer hardware, a parallel algorithm represents a significant advantage over alternative serial approaches.

7.2.2 Connections to hippocampal-cortical circuit

Some parallels can be drawn between the structure of the learning model presented in this thesis and the structure of the hippocampal-cortical circuit, which may perform a similar function to the model (see Figure 7.1). Based on an extensive body of experimental evidence (e.g., Squire (1982)) it is generally accepted that the hippocampal formation has a pivotal role in human learning and in the formation and retrieval of declarative memories (memories of explicit events). The hippocampus receives inputs from across the cortex, including all sensory areas, and so its inputs effectively represent the current state of the world, as represented in the cortex. Outputs from the hippocampus are also widely distributed to both cortical areas and the cerebellum.

In particular, it has been proposed that the conjunctive, predictive representations formed in the hippocampus are available to other learning systems in the brain (e.g., to cerebellum and cortex) (Gluck et al., 2005). For instance, it has been widely conjectured that the hippocampus participates in immediate, fast learning based on moment-to-moment events, with the learnt representations subsequently supporting long-term memory consolidation involving slower learning processes sited in the cortex (McClelland et al., 1995). This dichotomy is broadly similar to the split between the voted perceptrons building implicit action models from immediate experience, and the later extraction of explicit rules from the resulting representation. Similarly, hippocampal

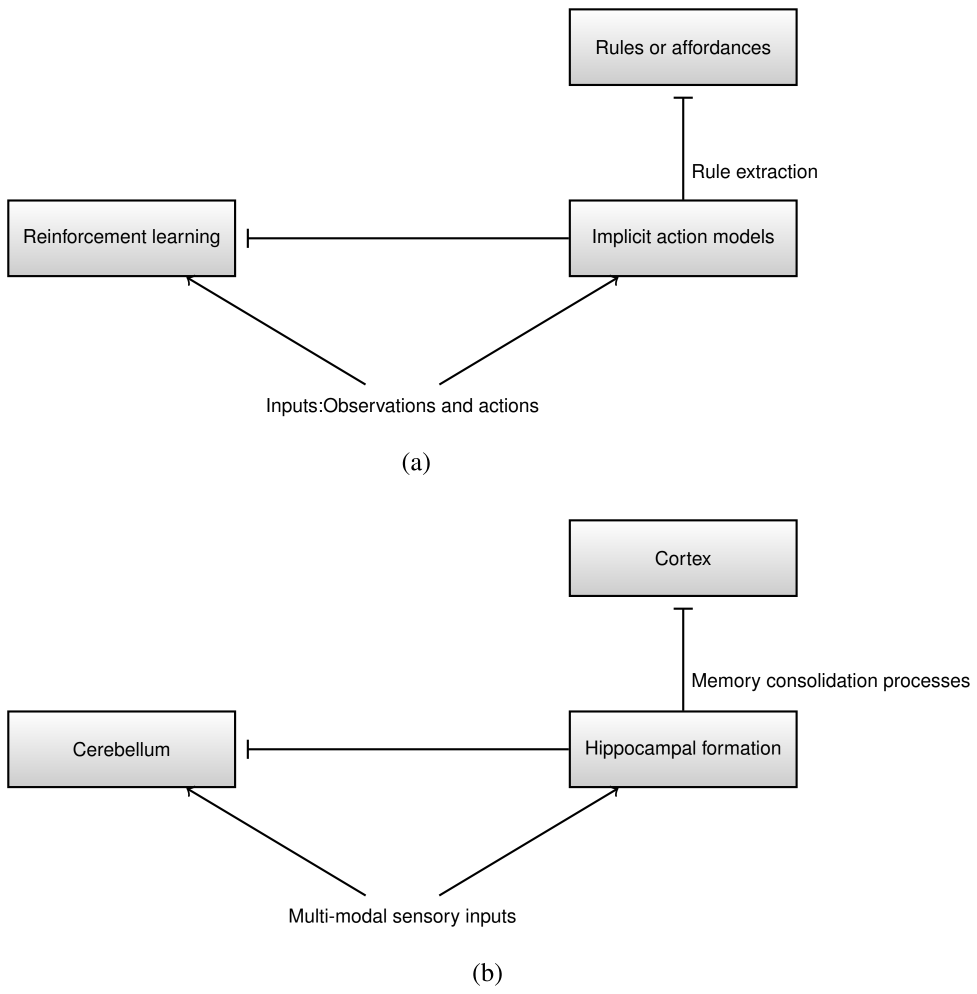


Figure 7.1: Comparison between the structure of the hippocampal-cortical circuit and the learning model. (a) The structure of the learning model. (b) The structure of the hippocampal-cortical circuit (adapted from Gluck et al. (2005), and Steedman (2004)).

representations appear to support cerebellar reinforcement learning, while it has also been noted in earlier chapters that the representations learnt by the voted perceptron models could be used in reinforcement learning.

Moreover, there are correspondences between the functionality of components of the learning model and postulated functionality of the main components of the hippocampus. Within the hippocampus, the dentate gyrus acts on its input representations to produce sparse, conjunctive representations, which differentiate between different contexts or outcomes. This is similar to the action of a kernel function, a correspondence which has been previously noted (Baker, 2003). The hippocampal subfield CA3 is thought to perform an autoassociative function, because there are many recurrent widely distributed projections within the subfield (Norman and O'Reilly, 2003). A related proposal is that the recurrency in CA3 supports sequence learning and temporal processing where previous and current contexts (or current and predicted contexts) are compared (Levy, 1996). The CA1 subfield may have a similar role but over longer time intervals. The voted perceptron model needs similar functionality, to be able to calculate deltas — differences between the current and previous states — and also to compare the current state with support vectors, that is, previously stored states. The voted perceptron's requirement for access to previously stored states also coincides with the broad function of the hippocampus as a store for episodic memories.

Thus potentially the hippocampus has available to it the basis for storing observed states (or support vectors), a function which compares consecutive states (calculates deltas), a function which sparsifies inputs (part of the function of a kernel), and a function which compares temporally distant states (also part of the function of a kernel). Additionally, the hippocampus is implicated in the learning of action models by humans.

The similarities between the learning model and the hippocampal-cortical circuit suggest that it may be possible to adapt existing models of the hippocampus to perform the task of learning action models, by basing a transformation on the structure of the model presented in this thesis. Such an adaptation could have a number of benefits, providing an alternative biologically plausible model of action learning, giving insight into how the hippocampus itself might perform action learning, and into how deictic references might be realised in the hippocampus.

7.2.3 The grounding gap

Learning grounded action models depends upon the existence of grounded relations, yet a process for grounding relations in sensorimotor experience is so far just beginning to be addressed in the research literature (Pierce and Kuipers, 1997; Modayil and Kuipers, 2008; Mugan, 2010). This grounding gap needs to be resolved in order to fully ground action models.

One approach is suggested by the two different definitions of an object which underlie the representations used in this thesis. In terms of affordances, an object is defined as the set of actions which it affords in a given context. In terms of deictic references, an object is defined as the set of relations which it participates in, in a given context. Equating the two definitions suggests that a set of relations is in some way equivalent to a set of actions. Relations could be considered to encode potential actions (and therefore also previous actions). For instance, an object may be "on" a table because specific actions have placed it there, or because specific actions may cause it to no longer be there. Since affordances are by definition grounded in sensorimotor experience it follows that relations are also embodied and can be grounded in sensorimotor experience, in contrast to attempts to ground relations in the visual perception of physical regularities of the world (e.g., Regier and Carlson (2001)).

In recent work Vankov (2010) makes a very similar proposal, that relations are grounded in action. However, the actions considered are those involved in visual attention, namely saccades and head movements. Moreover Vankov does not demonstrate how to acquire grounded relations, but only shows how pre-defined spatial relations may be represented in terms of (attentional) actions.

An important step for future work is therefore to consider the possibility that relations may be grounded in sensorimotor experience. Regardless of its exact nature, the grounding of relations is essential for true autonomous learning of domain dynamics, and so is a critical future direction.

Appendix A

This appendix contains PDDL files describing BlocksWorld, ZenoTravel, Depots, DriverLog, Briefcase, Elevator and Rover domains.

```
(define (domain blocksworld)
  (:requirements :strips)
  (:predicates (arm-empty)
    (clear ?x)
    (ontable ?x)
    (holding ?x)
    (on ?x ?y))

(:action pickup
  :parameters (?ob)
  :precondition  (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob)) (not (arm-empty))))

(:action putdown
  :parameters (?ob)
  :precondition (holding ?ob)
  :effect ((and (clear ?ob) (arm-empty) (on-table ?ob) (not (holding ?ob))))

(:action stack
  :parameters (?ob ?underob)
  :precondition (and (clear ?underob) (holding ?ob))
  :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob) (not (clear ?underob))
          (not (holding ?ob))))

(:action unstack
  :parameters (?ob ?underob)
  :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
  :effect (and (holding ?ob) (clear ?underob) (not (on ?ob ?underob)) (not (clear ?ob))
          (not (arm-empty)))))
```

Figure A.1: PDDL description of the BlocksWorld domain.

```
(define (domain briefcase)
  (:requirements :adl)
  (:types portable location)
  (:predicates (at ?y - portable ?x - location)
               (in ?x - portable)
               (is-at ?x - location))

(:action move
  :parameters (?m ?l - location)
  :precondition  (is-at ?m)
  :effect (and (is-at ?l) (not (is-at ?m))
      (forall (?x - portable) (when (in ?x)
        (and (at ?x ?l) (not (at ?x ?m)))))))

(:action take-out
  :parameters (?x - portable)
  :precondition (in ?x)
  :effect (not (in ?x)))

(:action put-in
  :parameters (?x - portable ?l - location)
  :precondition (and (not (in ?x)) (at ?x ?l) (is-at ?l))
  :effect (in ?x)))

}
```

Figure A.2: PDDL description of the Briefcase domain.

```
 (define (domain zeno-travel)
  (:requirements :typing)
  (:types aircraft person city flevel - object)
  (:predicates (at ?x - (either person aircraft) ?c - city)
   (in ?p - person ?a - aircraft)
   (fuel-level ?a - aircraft ?l - flevel)
   (next ?l1 ?l2 - flevel))


(:action board
  :parameters (?p - person ?a - aircraft ?c - city)
  :precondition (and (at ?p - person ?c - city) (at ?a - aircraft ?c - city))
  :effect (and (not (at ?p - person ?c - city)) (in ?p - person ?a - aircraft)))

(:action debark
  :parameters (?p - person ?a - aircraft ?c - city)
  :precondition (and (in ?p - person ?a - aircraft) (at ?a - aircraft ?c - city))
  :effect (and (not (in ?p - person ?a - aircraft)) (at ?p - person ?c - city)))

(:action fly
  :parameters (?a - aircraft ?c1 - city ?c2 - city ?l1 - flevel ?l2 - flevel)
  :precondition (and (at ?a - aircraft ?c1 - city) (fuel-level ?a - aircraft ?l1 - flevel)
                   (next ?l2 - flevel ?l1 - flevel))
  :effect (and (not (at ?a - aircraft ?c1 - city)) (at ?a - aircraft ?c2 - city)
              (not (fuel-level ?a - aircraft ?l1 - flevel))
                   (fuel-level ?a - aircraft ?l2 - flevel)))

(:action zoom
  :parameters (?a - aircraft ?c1 - city ?c2 - city ?l1 - flevel ?l2 - flevel ?l3 - flevel)
  :precondition (and (at ?a - aircraft ?c1 - city) (fuel-level ?a - aircraft ?l1 - flevel)
                   (next ?l2 - flevel ?l1 - flevel) (next ?l3 - flevel ?l2 - flevel))
  :effect (and (not (at ?a - aircraft ?c1 - city)) (at ?a - aircraft ?c2 - city)
              (not (fuel-level ?a - aircraft ?l1 - flevel))
                   (fuel-level ?a - aircraft ?l3 - flevel)))

(:action refuel
  :parameters (?a - aircraft ?c - city ?l - flevel ?l1 - flevel)
  :precondition (and (fuel-level ?a - aircraft ?l - flevel) (next ?l - flevel ?l1 - flevel)
                   (at ?a - aircraft ?c - city))
  :effect (and (fuel-level ?a - aircraft ?l1 - flevel)
              (not (fuel-level ?a - aircraft ?l - flevel)))))
```

Figure A.3: PDDL description of the ZenoTravel domain.

```
(define (domain Depot)
 (:requirements :typing)
 (:types place locatable - object
  depot distributor - place
  truck hoist surface - locatable
  pallet crate - surface)

 (:predicates (at ?x - locatable ?y - place)
  (on ?x - crate ?y - surface)
  (in ?x - crate ?y - truck)
  (lifting ?x - hoist ?y - crate)
  (available ?x - hoist)
  (clear ?x - surface))

(:action Drive
     :parameters (?x - truck ?y - place ?z - place)
     :precondition (and (at ?x ?y))
     :effect (and (not (at ?x ?y)) (at ?x ?z)))

(:action Lift
     :parameters (?x - hoist ?y - crate ?z - surface ?p - place)
     :precondition (and (at ?x ?p) (available ?x) (at ?y ?p) (on ?y ?z) (clear ?y))
     :effect (and (not (at ?y ?p)) (lifting ?x ?y) (not (clear ?y)) (not (available ?x)) (clear ?z)
                 (not (on ?y ?z))))

(:action Drop
     :parameters (?x - hoist ?y - crate ?z - surface ?p - place)
     :precondition (and (at ?x ?p) (at ?z ?p) (clear ?z) (lifting ?x ?y))
     :effect (and (available ?x) (not (lifting ?x ?y)) (at ?y ?p) (not (clear ?z)) (clear ?y)
   (on ?y ?z)))

(:action Load
     :parameters (?x - hoist ?y - crate ?z - truck ?p - place)
     :precondition (and (at ?x ?p) (at ?z ?p) (lifting ?x ?y))
     :effect (and (not (lifting ?x ?y)) (in ?y ?z) (available ?x)))

(:action Unload
     :parameters (?x - hoist ?y - crate ?z - truck ?p - place)
     :precondition (and (at ?x ?p) (at ?z ?p) (available ?x) (in ?y ?z))
     :effect (and (not (in ?y ?z)) (not (available ?x)) (lifting ?x ?y))))
```

Figure A.4: PDDL description of the Depots domain.

```
 (define (domain driverlog)
  (:requirements :typing)
  (:types location locatable - object
   driver truck obj - locatable)
  (:predicates (at ?obj - locatable ?loc - location)
   (in ?obj1 - obj ?obj - truck)
   (driving ?d - driver ?v - truck)
   (link ?x ?y - location) (path ?x ?y - location)
   (empty ?v - truck))

 (:action LOAD-TRUCK
      :parameters (?obj - obj ?truck - truck ?loc - location)
      :precondition (and (at ?truck ?loc) (at ?obj ?loc))
      :effect (and (not (at ?obj ?loc)) (in ?obj ?truck)))

 (:action UNLOAD-TRUCK
      :parameters (?obj - obj ?truck - truck ?loc - location)
      :precondition (and (at ?truck ?loc) (in ?obj ?truck))
      :effect (and (not (in ?obj ?truck)) (at ?obj ?loc)))

 (:action BOARD-TRUCK
      :parameters (?driver - driver ?truck - truck ?loc - location)
      :precondition (and (at ?truck ?loc) (at ?driver ?loc) (empty ?truck))
      :effect (and (not (at ?driver ?loc)) (driving ?driver ?truck) (not (empty ?truck))))

 (:action DISEMBARK-TRUCK
      :parameters (?driver - driver ?truck - truck ?loc - location)
      :precondition (and (at ?truck ?loc) (driving ?driver ?truck))
      :effect (and (not (driving ?driver ?truck)) (at ?driver ?loc) (empty ?truck)))

 (:action DRIVE-TRUCK
      :parameters (?truck - truck ?loc-from - location ?loc-to - location ?driver - driver)
      :precondition (and (at ?truck ?loc-from) (driving ?driver ?truck) (link ?loc-from ?loc-to))
      :effect (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))

(:action WALK
      :parameters (?driver - driver ?loc-from - location ?loc-to - location)
      :precondition (and (at ?driver ?loc-from) (path ?loc-from ?loc-to))
      :effect (and (not (at ?driver ?loc-from)) (at ?driver ?loc-to))))
```

Figure A.5: PDDL description of the DriverLog domain.

```
(define (domain miconic)
  (:requirements :adl :typing)
  (:types passenger - object
          floor - object)

(:predicates
  (origin ?person - passenger ?floor - floor)
  (destin ?person - passenger ?floor - floor)
  (above ?floor1 - floor  ?floor2 - floor)
  (boarded ?person - passenger)
  (served ?person - passenger)
  (lift-at ?floor - floor))



(:action stop
  :parameters (?f - floor)
  :precondition (lift-at ?f)
  :effect (and
               (forall (?p - passenger)
                  (when (and (boarded ?p)
                             (destin ?p ?f))
                        (and (not (boarded ?p))
                             (served  ?p))))
               (forall (?p - passenger)
                   (when (and (origin ?p ?f) (not (served ?p)))
                              (boarded ?p)))))


(:action up
  :parameters (?f1 - floor ?f2 - floor)
  :precondition (and (lift-at ?f1) (above ?f1 ?f2))
  :effect (and (lift-at ?f2) (not (lift-at ?f1))))

(:action down
  :parameters (?f1 - floor ?f2 - floor)
  :precondition (and (lift-at ?f1) (above ?f2 ?f1))
  :effect (and (lift-at ?f2) (not (lift-at ?f1))))
)
```

Figure A.6: PDDL description of the Elevator domain.

```
(define (domain Rover)
(:requirements :typing)
(:types rover waypoint store camera mode lander objective)

(:predicates (at ?x - rover ?y - waypoint)
             (at_lander ?x - lander ?y - waypoint)
             (can_traverse ?r - rover ?x - waypoint ?y - waypoint)
             (equipped_for_soil_analysis ?r - rover)
             (equipped_for_rock_analysis ?r - rover)
             (equipped_for_imaging ?r - rover)
             (empty ?s - store)
             (have_rock_analysis ?r - rover ?w - waypoint)
             (have_soil_analysis ?r - rover ?w - waypoint)
             (full ?s - store)
             (calibrated ?c - camera ?r - rover)
             (supports ?c - camera ?m - mode)
             (available ?r - rover)
             (visible ?w - waypoint ?p - waypoint)
             (have_image ?r - rover ?o - objective ?m - mode)
             (communicated_soil_data ?w - waypoint)
             (communicated_rock_data ?w - waypoint)
             (communicated_image_data ?o - objective ?m - mode)
             (at_soil_sample ?w - waypoint)
             (at_rock_sample ?w - waypoint)
             (visible_from ?o - objective ?w - waypoint)
             (store_of ?s - store ?r - rover)
             (calibration_target ?i - camera ?o - objective)
             (on_board ?i - camera ?r - rover)
             (channel_free ?l - lander))
```

Figure A.7: PDDL description of the Rover domain (preamble).

```
(:action navigate
:parameters (?x - rover ?y - waypoint ?z - waypoint)
:precondition (and (can_traverse ?x ?y ?z) (available ?x) (at ?x ?y) (visible ?y ?z))
:effect (and (not (at ?x ?y)) (at ?x ?z)))

(:action sample_soil
:parameters (?x - rover ?s - store ?p - waypoint)
:precondition (and (at ?x ?p) (at_soil_sample ?p) (equipped_for_soil_analysis ?x) (store_of ?s ?x)
                   (empty ?s))
:effect (and (not (empty ?s)) (full ?s) (have_soil_analysis ?x ?p) (not (at_soil_sample ?p))))

(:action sample_rock
:parameters (?x - rover ?s - store ?p - waypoint)
:precondition (and (at ?x ?p) (at_rock_sample ?p) (equipped_for_rock_analysis ?x) (store_of ?s ?x)
                   (empty ?s))
:effect (and (not (empty ?s)) (full ?s) (have_rock_analysis ?x ?p) (not (at_rock_sample ?p))))

(:action drop
:parameters (?x - rover ?y - store)
:precondition (and (store_of ?y ?x) (full ?y))
:effect (and (not (full ?y)) (empty ?y)))

(:action calibrate
 :parameters (?r - rover ?i - camera ?t - objective ?w - waypoint)
 :precondition (and (equipped_for_imaging ?r) (calibration_target ?i ?t) (at ?r ?w)
                    (visible_from ?t ?w)(on_board ?i ?r))
 :effect (calibrated ?i ?r) )

(:action take_image
 :parameters (?r - rover ?p - waypoint ?o - objective ?i - camera ?m - mode)
 :precondition (and (calibrated ?i ?r) (on_board ?i ?r) (equipped_for_imaging ?r) (supports ?i ?m)
                    (visible_from ?o ?p) (at ?r ?p))
 :effect (and (have_image ?r ?o ?m)(not (calibrated ?i ?r))))

(:action communicate_soil_data
 :parameters (?r - rover ?l - lander ?p - waypoint ?x - waypoint ?y - waypoint)
 :precondition (and (at ?r ?x)(at_lander ?l ?y)(have_soil_analysis ?r ?p)
                   (visible ?x ?y)(available ?r)(channel_free ?l))
 :effect (and (not (available ?r))(not (channel_free ?l))(channel_free ?l)
              (communicated_soil_data ?p)(available ?r)))

(:action communicate_rock_data
 :parameters (?r - rover ?l - lander ?p - waypoint ?x - waypoint ?y - waypoint)
 :precondition (and (at ?r ?x)(at_lander ?l ?y)(have_rock_analysis ?r ?p)
                    (visible ?x ?y)(available ?r)(channel_free ?l))
 :effect (and (not (available ?r))(not (channel_free ?l))(channel_free ?l)
              (communicated_rock_data ?p)(available ?r)))

(:action communicate_image_data
 :parameters (?r - rover ?l - lander ?o - objective ?m - mode ?x - waypoint ?y - waypoint)
 :precondition (and (at ?r ?x)(at_lander ?l ?y)(have_image ?r ?o ?m)(visible ?x ?y)
                    (available ?r)(channel_free ?l))
 :effect (and (not (available ?r))(not (channel_free ?l))(channel_free ?l)
              (communicated_image_data ?o ?m)(available ?r))))
```

Figure A.8: PDDL description of the Rover domain (continued).

Bibliography

Aggarwal, J. K., Davis, L. S., and Martin, W. N. (1981). Correspondence processes in dynamic scene analysis. *Proceedings of the IEEE*, 69(5):562–572.

Agre, P. E. and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI 1987)*, pages 268–272.

Amir, E. and Chang, A. (2008). Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, 33:349–402.

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, 2(4):319–342.

Başeski, E., Kraft, D., and Krüger, N. (2009). A hierarchical 3D circle detection algorithm applied in a grasping scenario. In *Proceedings of the International Conference on Computer Vision Theory and Applications (VISAPP 2009)*, pages 497–502.

Baker, J. L. (2003). Is there a support vector machine hiding in the dentate gyrus? *Neurocomputing*, 52-54:199–207.

Ballard, D. H., Hayhoe, M. M., Pook, P. K., and Rao, R. P. (1997). Deictic codes for the embodiment of cognition. *Behavioral and Brain Sciences*, 20(4):723–742.

Barakat, N. and Bradley, A. P. (2010). Rule extraction from support vector machines: A review. *Neurocomputing*, 74(1-3):178–190.

Barakat, N. H. and Bradley, A. P. (2007). Rule extraction from support vector machines: A sequential covering approach. *IEEE Transactions on Knowledge and Data Engineering*, 19(6):729–741.

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

Block, H. (1962). The perceptron: A model for brain functioning. *Reviews of Modern Physics*, 34:123–135.

Borgwardt, K. M. and Kriegel, H.-P. (2005). Shortest-path kernels on graphs. In *Proceedings of the 5th IEEE International Conference on Data Mining*, pages 74–81.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory (COLT 1992)*, pages 144–152.

Bouthinon, D., Soldano, H., and Ventos, V. (2009). Concept learning from (very) ambiguous examples. In *Proceedings of the 6th International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM 2009)*, pages 465–478.

Boutilier, C., Dearden, R., and Goldszmidt, M. (1995). Exploiting structure in policy construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 1995)*, pages 1104–1111.

Boutilier, C. and Poole, D. (1996). Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI 1996)*, volume 2, pages 1168–1175.

Boutilier, C., Reiter, R., Soutchanski, M., and Thrun, S. (2000). Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 355–362.

Brafman, R. I. and Tennenholtz, M. (2002). R-max — a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231.

Brooks, R. A. (1990). Elephants don't play chess. *Robotics and Autonomous Systems*, 6(1&2):3–15.

Bullot, N. and Droulez, J. (2008). Keeping track of invisible individuals while exploring a spatial layout with partial cues: Location-based and deictic direction-based strategies. *Philosophical Psychology*, 21(1):15–46.

Burgard, W., Fox, D., Hennig, D., and Schmidt, T. (1996). Estimating the absolute position of a mobile robot using position probability grids. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI 1996) Volume 2*, pages 896–901.

Cai, C., Liao, X., and Carin, L. (2009). Learning to explore and exploit in POMDPs. In *Advances in Neural Information Processing Systems (NIPS 22)*, pages 198–206.

Chakraborty, D. and Stone, P. (2011). Structure learning in ergodic factored MDPs without knowledge of the transition function's in-degree. In *Proceedings of the 28th International Conference on Machine Learning (ICML 2011)*, pages 737–744.

Chen, Z., Li, J., and Wei, L. (2007). A multiple kernel support vector machine scheme for feature selection and rule extraction from gene expression data of cancer tissue. *Artificial Intelligence in Medicine*, 41(2):161–175.

Collins, M. (2002). Discriminative training methods for Hidden Markov Models: Theory and experiments with perceptron algorithms. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP 2009)*, pages 1–8.

Collins, M. and Duffy, N. (2002). New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL 2002)*, pages 263–270.

Cresswell, S. and Gregory, P. (2011). Generalised domain model acquisition from action traces. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS 2011)*, pages 42–49.

Croonenborghs, T., Ramon, J., Blockeel, H., and Bruynooghe, M. (2007). Online learning and exploiting relational models in reinforcement learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 726–731.

Şahin, E., Çakmak, M., Doğar, M. R., Uğur, E., and Üçoluk, G. (2007). To afford or not to afford: A new formalization of affordances towards affordance based robot control. *Adaptive Behavior*, 15(4):447–472.

Dabney, W. and McGovern, A. (2006). The thing we tried that worked: Utile distinctions for relational reinforcement learning. In *Proceedings of the ICML Workshop on Open Problems in Statistical Relational Learning (SRL 2006)*.

De Raedt, L. (1997). Logical settings for concept-learning. *Artificial Intelligence*, 95(1):187–201.

De Raedt, L. (2008). *Logical and Relational Learning (Cognitive Technologies)*. Springer, first edition.

De Raedt, L. and Džeroski, S. (1994). First-order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392.

Dean, T. and Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, 5(2):142–150.

Decatur, S. E. and Gennaro, R. (1995). On learning from noisy and incomplete examples. In *Proceedings of the 8th Annual Conference on Computational Learning Theory (COLT 1995)*, pages 353–360.

Diuk, C., Li, L., and Leffler, B. R. (2009). The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009)*, pages 249–256.

Doshi-Velez, F., Pineau, J., and Roy, N. (2012). Reinforcement learning with limited reinforcement: Using Bayes risk for active learning in POMDPs. *Artificial Intelligence*, 187-188:115–132.

Doğar, M. R., Çakmak, M., Uğur, E., and Şahin, E. (2007). From primitive behaviors to goal directed behavior using affordances. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS 2007)*, pages 729–734.

Driessens, K., Ramon, J., and Gärtner, T. (2006). Graph kernels and Gaussian processes for relational reinforcement learning. *Machine Learning*, 64(1):91–119.

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

Finegan, E. (1998). *Language: Its Structure and Use*. Heinle & Heinle Publishers, third edition.

Finger, J. J. (1987). *Exploiting constraints in design synthesis*. PhD thesis, Stanford University, Stanford, CA, USA.

Finney, S., Gardiol, N. H., Kaelbling, L. P., and Oates, T. (2002). The thing that we tried didn't work very well: Deictic representation in reinforcement learning. In *Proceedings of the 18th International Conference on Uncertainty in Artificial Intelligence (UAI 2002)*, pages 154–161.

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

Fu, L. (1991). Rule learning by searching on adapted nets. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI 1991)*, pages 590–595.

Fu, X., Ong, C., Keerthi, S., Hung, G. G., and Goh, L. (2004). Extracting the knowledge embedded in support vector machines. In *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN 2004)*, volume 1, pages 291–296.

Fung, G., Sandilya, S., and Rao, R. B. (2005). Rule extraction from linear support vector machines. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD 2005)*, pages 32–40.

Gardiol, N. H. and Kaelbling, L. P. (2007). Action-space partitioning for planning. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI 2007)*, pages 980–986.

Gärtner, T., Driessens, K., and Ramon, J. (2003a). Graph kernels and Gaussian processes for relational reinforcement learning. In *Proceedings of the International Conference on Inductive Logic Programming (ILP 2003)*, pages 146–163.

Gärtner, T., Flach, P., and Wrobel, S. (2003b). On graph kernels: Hardness results and efficient alternatives. In *Proceedings of the 16th Annual Conference on Computational Learning Theory and 7th Kernel Workshop (COLT 2003)*, pages 129–143.

Getoor, L., Friedman, N., Koller, D., Pfeffer, A., and Taskar, B. (2007). *Introduction to Statistical Relational Learning*, chapter 5. The MIT Press.

Gibson, J. J. (1979). *The Ecological Approach To Visual Perception*. Lawrence Erlbaum Associates.

Gil, Y. (1994). Learning by experimentation: Incremental refinement of incomplete planning domains. *Proceedings of the 11th International Conference on Machine Learning (ICML 1994)*, pages 87–95.

Gluck, M. A., Myers, C., and Meeter, M. (2005). Cortico-hippocampal interaction and adaptive stimulus representation: A neurocomputational theory of associative learning and memory. *Neural Networks*, 18(9):1265–1279.

Goodale, M. A. and Milner, A. D. (1992). Separate visual pathways for perception and action. *Trends in Neurosciences*, 15(1):20–25.

Graepel, T., Herbrich, R., and Williamson, R. C. (2000). From margin to sparsity. In *Advances in Neural Information Processing Systems (NIPS 13)*, pages 210–216.

Guestrin, C., Koller, D., Gearhart, C., and Kanodia, N. (2003). Generalizing plans to new environments in relational MDPs. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 1003–1010.

Guestrin, C., Patrascu, R., and Schuurmans, D. (2002). Algorithm-directed exploration for model-based reinforcement learning in factored MDPs. In *Proceedings of the 19th International Conference on Machine Learning (ICML 2002)*, pages 235–242.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182.

Guyon, I., Weston, J., Barnhill, S., and Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. *Machine Learning*, 46(1-3):389–422.

Halbritter, F. and Geibel, P. (2007). Learning models of relational MDPs using graph kernels. In *Proceedings of the 6th Mexican International Conference on Advances in Artificial Intelligence (MICAI 2007)*, pages 409–419.

Halford, G. S., Wilson, W. H., Phillips, S., Siegler, R., Flavell, J., Kotovsky, K., Mcclelland, J., Baddeley, A., Tsotsos, J., Shastri, L., and Berch, D. (1988). Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioral and Brain Sciences*, 21(6):803–864.

Harnad, S. (1990). The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1-3):335–346.

Haussler, D. (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36(2):177–221.

Haussler, D. (1989). Learning conjunctive concepts in structural domains. *Machine Learning*, 4(1):7–40.

Hayes, B., Curtiss, S., Szabolcsi, A., Stowell, T., Stabler, E., Sportiche, D., Koopman, H., Keating, P., Munro, P., Hyams, N., and Steriade, D. (2001). *Linguistics: An introduction to linguistic theory*. Wiley-Blackwell.

Holmes, M. and Isbell, C. (2005). Schema learning: Experience-based construction of predictive action models. In *Advances in Neural Information Processing Systems (NIPS 17)*, pages 585–562.

Hommel, B. (2004). Event files: Feature binding in and across perception and action. *Trends in Cognitive Sciences*, 8(11):494–500.

Hurford, J. R. (2003). The neural basis of predicate-argument structure. *Behavioral and Brain Sciences*, 23(6):261–283.

Itti, L., Koch, C., and Niebur, E. (1998). A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(11):1254–1259.

Jonsson, A. and Barto, A. (2007). Active learning of Dynamic Bayesian Networks in Markov Decision Processes. In *Proceedings of the 7th International Conference on Abstraction, Reformulation, and Approximation (SARA 2007)*, pages 273–284.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134.

Kahneman, D., Treisman, A., and Gibbs, B. J. (1992). The reviewing of object files: Object-specific integration of information. *Cognitive Psychology*, 24(2):175–219.

Kapur, D. and Narendran, P. (1986). NP-completeness of the set unification and matching problems. In *Proceedings of the 8th International Conference on Automated Deduction*, pages 489–495.

Kashima, H., Tsuda, K., and Inokuchi, A. (2003). Marginalized kernels between labeled graphs. In *Proceedings of the 20th International Conference on Machine Learning (ICML 2003)*, pages 321–328.

Kearns, M. and Koller, D. (1999). Efficient reinforcement learning in factored MDPs. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*, pages 740–747.

Kearns, M. and Singh, S. (1998). Near-optimal performance for reinforcement learning in polynomial time. In *Proceedings of the 15th International Conference on Machine Learning (ICML 1998)*, pages 260–268.

Kearns, M. and Singh, S. (2002). Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3):209–232.

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

Khardon, R. and Servedio, R. A. (2005). Maximum margin algorithms with Boolean kernels. *Journal of Machine Learning Research*, 6:1405–1429.

Khardon, R. and Wachman, G. M. (2007). Noise tolerant variants of the perceptron algorithm. *Journal of Machine Learning Research*, 8:227–248.

Klivans, A. R. and Servedio, R. A. (2004). Learning intersections of halfspaces with a margin. In *Proceedings of the 17th Annual Conference on Learning Theory (COLT 2004)*, pages 348–362.

Koch, C. and Ullman, S. (1985). Shifts in selective visual attention: Towards the underlying neural circuitry. *Human Neurobiology*, 4(4):219–227.

Kohavi, R. and John, G. H. (1997). Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324.

Kraft, D., Detry, R., Pugeault, N., Baseski, E., Guerin, F., Piater, J. H., and Kruger, N. (2010). Development of object and grasping knowledge by robot exploration. *IEEE Transactions on Autonomous Mental Development*, 2(4):368–383.

Krishnan, R., Sivakumar, G., and Bhattacharya, P. (1999). A search technique for rule extraction from trained neural networks. *Pattern Recognition Letters*, 20(3):273–280.

Krüger, N., Geib, C., Piater, J., Petrick, R., Steedman, M., Wörgötter, F., Ude, A., Asfour, T., Kraft, D., Omrčen, D., Agostini, A., and Dillmann, R. (2011). Object-Action Complexes: Grounded abstractions of sensorimotor processes. *Robotics and Autonomous Systems*, 59(10):740–757.

Laird, J. E. (2008). Extending the soar cognitive architecture. In *Proceedings of the 2008 Conference on Artificial General Intelligence*, pages 224–235.

Lang, T. (2011). *Planning and Exploration in Stochastic Relational Worlds*. PhD thesis, Freie Universität Berlin.

Lang, T. and Toussaint, M. (2010). Planning with noisy probabilistic relational rules. *Journal of Artificial Intelligence Research*, 39(1):1–49.

Lang, T., Toussaint, M., and Kersting, K. (2010). Exploration in relational worlds. In *Proceedings of the European Conference on Machine Learning (ECML PKDD 2010)*, pages 178–194.

Levy, W. B. (1996). A sequence predicting CA3 is a flexible associator that learns and uses context to solve hippocampal-like tasks. *Hippocampus*, 6(6):579–590.

Martens, D., Huysmans, J., Setiono, R., Vanthienen, J., and Baesens, B. (2008). Rule extraction from support vector machines: An overview of issues and application in credit scoring. In *Studies in Computational Intelligence (SCI)*, pages 33–63.

McCarthy, J. (1977). Epistemological problems of artificial intelligence. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI 1977)*, volume 2, pages 1038–1044.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of Artificial Intelligence. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press.

McClelland, J. L., McNaughton, B. L., and O'Reilly, R. C. (1995). Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory. *Psychological Review*, 102(3):419–457.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Metta, G. and Fitzpatrick, P. (2003). Early integration of vision and manipulation. *Adaptive Behavior*, 11(2):109–128.

Michael, L. (2007). Learning from partial observations. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 968–974.

Minsky, M. L. and Papert, S. A. (1969). *Perceptrons*. The MIT Press.

Mitchell, T. (1982). Generalization as search. *Artificial Intelligence*, 18(2):203–226.

Modayil, J. and Kuipers, B. (2007). Where do actions come from? Autonomous robot learning of objects and actions. In *Proceedings of the AAAI 2007 Spring Symposium on Control Mechanisms for Spatial Knowledge Processing in Cognitive/Intelligent Systems*, pages 41–46.

Modayil, J. and Kuipers, B. (2008). The initial development of object knowledge by a learning robot. *Robotics and Autonomous Systems*, 56(11):879–890.

Modayil, J. and Kuipers, B. J. (2004). Bootstrap learning for object discovery. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*, volume 1, pages 742–747.

Montesano, L., Lopes, M., Bernardino, A., and Santos-Victor, J. (2008). Learning object affordances: From sensory–motor coordination to imitation. *IEEE Transactions on Robotics*, 24(1):15–26.

Mugan, J. (2010). *Autonomous Qualitative Learning of Distinctions and Actions in a Developing Agent*. PhD thesis, University of Texas at Austin.

Mukerjee, A. (2009). Using attentive focus to discover action ontologies from perception. In *Proceedings of the 5th International Workshop on Neural-Symbolic Learning and Reasoning (NeSy 2009)*.

Murphy, K. (2002). *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley.

Norman, K. A. and O'Reilly, R. C. (2003). Modeling hippocampal and neocortical contributions to recognition memory: a complementary-learning-systems approach. *Psychological Review*, 110(4):611–646.

Novikoff, A. B. (1963). On convergence proofs for perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622.

Núñez, H., Angulo, C., and Català, A. (2008). Rule extraction based on support and prototype vectors. In *Studies in Computational Intelligence (SCI)*, volume 80, pages 109–134. Springer.

Olsson, L. A., Nehaniv, C. L., and Polani, D. (2006). From unknown sensors and actuators to actions grounded in sensorimotor perceptions. *Connection Science*, 18(2):121–144.

O'Regan, J. K. (1992). Solving the "real" mysteries of visual perception: The world as an outside memory. *Canadian Journal of Psychology*, 46(3):461–488.

Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. (2007). Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352.

Pednault, E. P. D. (1989). ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332.

Peirce, C. S. (1931). *The collected papers of Charles Sanders Peirce*. Harvard University Press.

Petrick, R., Geib, C., and Steedman, M. (2010). Integrating low-level robot/vision with high-level planning and sensing in PACO-PLUS. Technical report, University of Edinburgh.

Petrick, R. P. A. and Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, pages 212–221.

Pierce, D. and Kuipers, B. J. (1997). Map learning with uninterpreted sensors and effectors. *Artificial Intelligence*, 92(1-2):169–227.

Plotkin, G. D. (1970). A note on inductive generalization. *Machine Intelligence*, 5:153–163.

Poole, D. (1997). The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):7–56.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience.

Pylyshyn, Z. (1989). The role of location indexes in spatial perception: A sketch of the FINST spatial-index model. *Cognition*, 32(1):65–97.

Pylyshyn, Z. W. (2000). Situating vision in the world. *Trends in Cognitive Sciences*, 4(5):197–207.

Pylyshyn, Z. W. (2001). Visual indexes, preconceptual objects, and situated vision. *Cognition*, 80(1-2):127–158.

Pylyshyn, Z. W. (2009). Perception, representation and the world: The FINST that binds. In Dedrick, D. and Trick, L. M., editors, *Computation, Cognition and Pylyshyn*, pages 3–48. MIT Press.

Rakotomamonjy, A. (2003). Variable selection using SVM-based criteria. *Journal of Machine Learning Research*, 3:1357–1370.

Ramon, J. and Gärtner, T. (2003). Expressivity versus efficiency of graph kernels. In *Proceedings of the First International Workshop on Mining Graphs, Trees and Sequences*, pages 65–74.

Regier, T. and Carlson, L. A. (2001). Grounding spatial language in perception: An empirical and computational investigation. *Journal of Experimental Psychology: General*, 130(2):273–298.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.

Rodrigues, C., Gérard, P., and Rouveirol, C. (2010a). Incremental learning of relational action models in noisy environments. In *Proceedings of the International Conference on Inductive Logic Programming (ILP 2010)*, pages 206–213.

Rodrigues, C., Gérard, P., Rouveirol, C., and Soldano, H. (2010b). Incremental learning of relational action rules. In *International Conference on Machine Learning and Applications (ICMLA 2010)*, pages 451–458.

Rodrigues, C., Gérard, P., Rouveirol, C., and Soldano, H. (2011). Active learning of relational action models. In *Proceedings of the 21st International Conference on Inductive Logic Programming (ILP 2011)*, pages 302–316.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Rosman, B. and Ramamoorthy, S. (2011). Learning spatial relationships between objects. *International Journal of Robotics Research*, 30(11):1328–1342.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, third edition.

Sadohara, K. (2001). Learning of Boolean functions using support vector machines. In *Proceedings of the 12th International Conference on Algorithmic Learning Theory (ALT 2001)*, pages 106–118.

Sadohara, K. (2002). On a capacity control using Boolean kernels for the learning of Boolean functions. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 410–417.

Safaei, J. and Sani, G. G. (2007). Incremental learning of planning operators in stochastic domains. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007)*, pages 644–655.

Sanner, S. and Kersting, K. (2010). Symbolic dynamic programming for first-order POMDPs. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*, pages 1140–1146.

Schmill, M. D., Oates, T., and Cohen, P. R. (2000). Learning planning operators in real-world, partially observable environments. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 246–253.

Schuurmans, D. and Greiner, R. (1997). Learning to classify incomplete examples. In *Computational Learning Theory and Natural Learning Systems: Volume IV: Making learning systems practical*, pages 87–105.

Schwind, C. (1999). Causality in action theories. *Electronic Transactions on Artificial Intelligence*, 3:27–50.

Setiono, R. (2000). Extracting M-of-N rules from trained neural networks. *IEEE Transactions on Neural Networks*, 11(2):512–519.

Shahaf, D. and Amir, E. (2006). Learning partially observable action schemas. In *Proceedings of the National Conference on Artificial Intelligence (AAAI 2006)*, pages 913–919.

Shani, G., Brafman, R. I., and Shimony, S. E. (2005). Model-based online learning of POMDPs. In *Proceedings of the 16th European Conference on Machine Learning (ECML 2005)*, pages 353–364.

Shervashidze, N. and Borgwardt, K. M. (2009). Fast subtree kernels on graphs. In *Advances in Neural Information Processing Systems (NIPS 22)*, pages 1660–1668.

Shervashidze, N., Vishwanathan, S. V. N., Petri, T., Mehlhorn, K., and Borgwardt, K. M. (2009). Efficient graphlet kernels for large graph comparison. *Journal of Machine Learning Research - Proceedings Track*, 5:488–495.

Singer, Y. and Crammer, K. (2003). Ultraconservative online algorithms for multiclass problems. *Journal of Machine Learning Research*, 3:951–991.

Sjöö, K. and Jensfelt, P. (2011). Learning spatial relations from functional simulation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, pages 1513–1519.

Slaney, J. and Thiébaux, S. (2001). Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153.

Slowinski, W. and Guerin, F. (2011). Learning regions for building a world model from clusters in probability distributions. In *Proceedings of the IEEE International Conference on Development and Learning (ICDL 2011)*, volume 2, pages 1–6.

Squire, L. R. (1982). The neuropsychology of human memory. *Annual Review of Neuroscience*, 5(1):241–273.

Steedman, M. (2002). Plans, affordances, and combinatory grammar. *Linguistics and Philosophy*, 25:723–753.

Steedman, M. (2004). Where does compositionality come from? In Levy, S. and Gayler, R., editors, *Proceedings of the AAAI Fall Symposium on Compositional Connectionism in Cognitive Science*, pages 59–62.

Strehl, A. L., Diuk, C., and Littman, M. L. (2007). Efficient structure learning in factored-state MDPs. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI 2007)*, volume 1, pages 645–650.

Surdeanu, M. and Ciaramita, M. (2007). Robust information extraction with perceptrons. In *Proceedings of the NIST 2007 Automatic Content Extraction Workshop (ACE07)*.

Sutton, R. S. (1990). Integrated architecture for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th International Conference on Machine Learning (ICML 1990)*, pages 216–224.

Teichteil-Königsbuch, F., Kuter, U., and Infantes, G. (2010). Incremental plan aggregation for generating policies in MDPs. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, volume 1, pages 1231–1238.

Tickle, A., Maire, F., Bologna, G., Andrews, R., and Diederich, J. (2000). Lessons from past, current issues, and future research directions in extracting the knowledge embedded in artificial neural networks. In *Hybrid Neural Systems*, pages 226–239.

Towell, G. G. and Shavlik, J. W. (1993). Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13(1):71–101.

Treisman, A. (1998). Feature binding, attention and object perception. *Philosophical Transactions of the Royal Society of London. Series B, Biological sciences*, 353(1373):1295–1306.

Uğur, E. and Şahin, E. (2010). Traversability: A case study for learning and perceiving affordances in robots. *Adaptive Behavior*, 18(3-4):258–284.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.

Valiant, L. G. (1985). Learning disjunctions of conjunctions. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI 1985)*, volume 1, pages 560–566.

Van Otterlo, M. (2009). *The Logic of Adaptive Behavior*. IOS Press.

Van Rijsbergen, C. J. (1979). *Information Retrieval*. Butterworth-Heinemann, 2nd edition.

Vankov, I. I. (2010). *Grounding relations and analogy-making in action*. PhD thesis, New Bulgarian University.

Wachman, G. and Khardon, R. (2007). Learning from interpretations: A rooted kernel for ordered hypergraphs. In *Proceedings of the 24th International Conference on Machine Learning (ICML 2007)*, pages 943–950.

Waldinger, R. J. (1977). Achieving several goals simultaneously. In Elcock, E. and Michie, D., editors, *Machine Intelligence 8*, pages 91–136. Ellis Horwood, Ltd.

Walsh, T. J. (2010). *Efficient Learning of Relational Models for Sequential Decision Making*. PhD thesis, Rutgers, The State University of New Jersey.

Walsh, T. J. and Littman, M. L. (2008). Efficient learning of action schemas and web-service descriptions. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI 2008)*, pages 714–719.

Wang, C. and Khardon, R. (2010). Relational partially observable MDPs. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*, pages 1153–1158.

Wang, X. (1995). Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the 12th International Conference on Machine Learning (ICML 1995)*, pages 549–557.

Weston, J., Mukherjee, S., Chapelle, O., Pontil, M., and Vapnik, V. (2000). Feature selection for SVMs. In *Advances in Neural Information Processing Systems (NIPS 13)*, pages 668–674.

Whitehead, S. D. and Ballard, D. H. (1991). Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83.

Xu, J. Z. and Laird, J. E. (2010). Instance-based online learning of deterministic relational action models. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*, pages 1574–1579.

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

Yoon, S. and Kambhampati, S. (2007). Towards model-lite planning: A proposal for learning and planning with incomplete domain models. In *ICAPS 2007: Workshop on AI Planning and Learning*.

Zhang, Y., Li, Z., and Cui, K. (2005a). DRC-BK: Mining classification rules by using Boolean kernels. In *Proceedings of the International Conference on Computational Science and Its Applications (ICCSA 2005)*, pages 214–222.

Zhang, Y., Li, Z., Tang, Y., and Cui, K. (2004). DRC-BK: Mining classification rules with help of SVM. In *Proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2004)*, pages 191–195.

Zhang, Y., Su, H., Jia, T., and Chu, J. (2005b). Rule extraction from trained support vector machines. In *Proceedings of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2005)*, pages 92–95.

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