Child, Christopher H. T. (2011). Approximate Dynamic Programming with Parallel Stochastic Planning Operators. (Unpublished Doctoral thesis, City University London)

CITY UNIVERSITY LONDON

City Research Online

City University London
Department of Computing

Approximate Dynamic Programming with Parallel Stochastic Planning Operators

Christopher H. T. Child

A thesis submitted for the degree of
Doctor of Philosophy at City University London

July 2011

Contents

List of Figures

List of Tables

List of Algorithms

ACHNOWLEDGEMENTS:

I'd like to thank my tutor, Dr. Kostas Stathis, for helping me to find both a fascinating research area and a range of techniques to explore within it. Our meetings have been exactly as academic research should be: inspiring, motivating and resulting in the phrase "needs further research". I have also been fortunate in having a second tutor Dr. Artur Garcez who provided the motivation to help me finish the thesis, guidance on analysis techniques, and the understanding that the life of a part-time student requires a subtle mix of carrot and stick. I'd also like to thank Dr. Andrew Tuson for introducing me to City University and helping to further my career as an academic. Thank you to all the staff of the School of Informatics at City University for your continued support and assistance.

I'd like to thank my wife, Cecil, for her inexhaustible patience and being the world's best girl. I couldn't have done this without you. Thank you to all the friends and house-mates who've helped encourage and, more importantly, keep me sane by distracting me. Finally, I'd like to thank my family for their incredible support, both financially and emotionally, without which I'd probably have given up and taken a job in banking.

Declaration

I hereby declare that:

- my submission as a whole is not substantially the same as any that I have previously made or am currently making, whether in published or unpublished form, for a degree, diploma, or similar qualification at any university or similar institution
- the following parts of the work or works now submitted have previously been submitted for a qualification at a university or similar institution (only brief details required):

  . . . . . .

  . . . . . .

  . . . . . .

- until the outcome of the current application to this University is known, the work or works submitted will not be submitted for any qualification at another university or similar institution.

Date: . . . . . Signature: . . . . . .

Print Name: . . . . . .

ABSTRACT:

This thesis presents an approximate dynamic programming (ADP) technique for environment modelling agents. The agent learns a set of parallel stochastic planning operators (P-SPOs) by evaluating changes in its environment in response to actions, using an association rule mining approach. An approximate policy is then derived by iteratively improving state value aggregation estimates attached to the operators using the P-SPOs as a model in a Dyna-Q-like architecture.

Reinforcement learning and dynamic programming are powerful techniques for automated agent decision making in stochastic environments. Dynamic programming is effective when there is a known environment model, while reinforcement learning is effective when a model is not available. The techniques derive a *policy*: a mapping from each environment state to an action which optimizes the long term reward the agent receives.

The standard methods become less effective as the state space for the environment increases because they require values to be associated with each state, the storage and processing of which is exponential to the number of state variables. Resolving this "curse of dimensionality" is an important topic of research amongst all communities working on this problem. Two key methods are to: (i) derive an estimate of the value (approximate dynamic programming) using function approximation or state aggregation; or (ii) build a model of the environment from experience.

This thesis presents a method of combining these approaches by exploiting structure in the state transition and value functions captured in a set of *planning operators* which are learnt through experience in the environment. Standard planning operators define the deterministic changes that occur in an environment in response to an action. This work presents Parallel Stochastic Planning Operators (P-SPOs), a novel form of planning operator providing a structured model of the state transition function in environments which are both non-deterministic and for which changes can occur outside the influence of actions. Next, an automated method for extracting P-SPOs from observations in an environment is explored using an adaptation of association rule mining. Finally, methods of relating the state transition structure encapsulated in the P-SPOs to state values, using the operators to store state value aggregation estimates, are evaluated.

The framework described provides a method by which approximate dynamic programming can be applied by designers of AI agents and AI planning systems for which they have minimal prior knowledge. The framework and P-SPO based implementations are tested against standard techniques in two bench-mark stochastic environments: a "slippery gripper" block painting robot; and a "predator-prey" agent environment.

Experimental results show that an agent using a P-SPO-based approach is able to learn an accurate model of its environment if successor state variables exhibit conditional independence, and an approximate model in the non-independent case. Results also demonstrate that the agent's ability to generalise to previously unseen states using the model allow it to form an improved policy over an agent employing a standard Dyna-Q based technique. Finally, an approximate policy stored in state aggregation estimates attached to operators is shown to be optimal in experiments for which the P-SPO set contains sufficient information for effective aggregations to be formed.

1. Introduction

It has been the aim of many AI researchers to create an autonomous agent that can be situated in an environment and learn to act effectively through discovery of the mechanics of the world they inhabit. This has been termed "developmental AI" [40] or "constructivist AI" [28][84] and is discussed as early as 1950 in Turing's paper "Computing, Machinery & Intelligence" [92] in which the idea of building a simulation of an infant's mind that could be trained through interaction with the world was proposed as one method of constructing a machine that could pass, what later became known as, the Turing test.

Turing's motivation for this aim was a practical one of solving issues with adaptability. It was clear that an artificial intelligence could not be programmed to respond to every eventuality that it could encounter, and that even if this knowledge could be given, it would soon become out of date as the environment changed. Some mechanism was therefore needed to adapt to the changing conditions and the development of knowledge by discovery promised an approach that required minimal programmer effort if appropriately general principles could be discovered.

This adaptability motivation is reflected in a number of agent-based applications, and is particularly apparent in the fields of adversarial AI and non-player character AI (NPC AI) in computer game applications. Computer games are played by humans, who continually adapt their strategies to improve their performance. If a weakness is found in an adversarial AI's behaviour, then the game will quickly become uninteresting if the AI opponent keeps re-playing the same losing strategy. A developmental approach could help the AI adapt to these changing strategies. NPC's in computer games can be adversaries (e.g. bots in FPS games), in which case the same argument applies, but they can also be helpers to the main character (e.g. a war-horse in a role-playing game), or simply background characters aimed at improving the aesthetics of the environment (e.g. a villager in a town the character travels through). Each of these agent types could benefit from a developmental approach that allows a designer to specify the type of behaviour that is required without having to specify the means of achieving it.

In this context an *autonomous agent* is considered to be a decision-making entity. It is situated in some *environment* or *world*, and has a number of *actions* that it can carry out. It has a method of perceiving its environment, and makes decisions as to which of the available actions it will select. It is autonomous in the sense that it can actively perform action selection without external intervention. The agent has a perceive function which converts sensor data

from the environment into a percept, and uses information contained within these percepts to guide its actions through some form of deliberation [49].

The type of agent studied in this research builds a model of its world by evaluating changes to the received percepts over time, and models the effects of its available actions by evaluating the changes in perception in response to the actions selected. The agent is given a reward mechanism, which indicates a preference that the agent should have for perceiving that it is in a particular state. The agent then uses its model to make decisions by forming a *plan* or *policy*. A plan is a deterministic set of actions, which lead the agent from its current state to a reward state. In stochastic (random) environments the agent cannot establish a deterministic set of actions and must, instead, create a strategy which takes into account every state it could find itself in. This strategy is called a policy (or universal plan), and is a mapping from every possible state to an action.

Planning in a stochastic environment in which actions have probabilistic outcomes, the environment changes outside the agent's control, or the agent has uncertain knowledge about the environment state, presents unique challenges which are not present in classical planning systems, such as STRIPS [31] or the situation calculus [57]. The random nature of the environment, from the agent's perspective, means it needs a mechanism for selecting action in situations or which are unexpected, or which it may not have encountered before. For large environments an exact definition of such a plan becomes impossible and approximation techniques are required. Such techniques fall under the categories of approximate dynamic programming (ADP) [73] and Decision Theoretic Planning [7].

The work presented here investigates the creation of an agent which:

- Builds a *planning operator* based model of its world through interaction. Planning operators describe the expected changes to the environment in response to the agent's actions.
- Uses the model, to attach *utility estimates* (estimates of expected future rewards) to the planning operators.
- Uses the *utility estimates* to provide a policy. The agent can select an action to activate the operators with the highest utility estimate. Given an initial percept, the agent can make a decision by finding the highest valued action available for that percept.

The syntax of the *planning operators* acquired by the agent will be covered in depth in chapter 5. In order to introduce the concept, a simple example of an operator set for an agent is given below.

The agent has two actions: *flip* or *doNothing* in an environment consisting of a single coin which can be showing either heads or tails. It receives a reward of 1.0 if the coin is showing heads and 0.0 otherwise:

$$flip(coin):\{\} \rightarrow \begin{Bmatrix} 0.5:showing(coin,heads) \\ 0.5:showing(coin,tails) \end{Bmatrix} U(0.5)$$
$$doNothing:showing(coin,heads) \rightarrow \{1.0:showing(coin,heads)\} U(1.0)$$
$$doNothing:showing(coin,tails) \rightarrow \{1.0:showing(coin,tails)\} U(0.0)$$

Each operator has:

- An action: e.g. *flip(coin).*
- A context: e.g. *showing(coin, heads).*
- An outcome set with associated probabilities. e.g. *{0.5:showing(coin, heads), 0.5:showing(coin, tails)}*
- A utility: e.g. *U(0.5).*

The outcome set identifies the expected changes to the environment in response to the action if the context holds. The utility is an estimate of the expected future rewards if the action is taken in the given context. The task in this case is *episodic* (has terminating states). The episode length is one, with both *showing(coin, heads)* and *showing(coin, tails)* being terminating stares. This means that only immediate rewards affect the utility.

The agent can form a policy by selecting the action with the best available utility in the given context. If, for example, the coin is currently showing heads, then the *flip* action can be taken (because it has no context) or the *doNothing* action can be taken (with *showing(coin, heads)* context). These have utilities of 0.5 and 1.0 respectively, and an agent attempting to maximise reward gather would, therefore, select the *doNothing* action, resulting in an immediate reward.

A method of learning operators of this type, along with their more complex parallel extensions, is defined in chapter 6, and evaluated in chapter 8. Methods of attaching utility estimates to planning operators are investigated in chapter 9, and evaluated in chapter 10.

Empirical learning of planning operators in stochastic environments is challenging because:

- An action may have uncertain effects inherently (e.g. the result of a “flip” action on a coin).

- The effects of an action may be masked by external elements (e.g. multiple coins are flipped simultaneously by others and the agent wrongly attributes the result of their others actions to its own).
- The action conditions may be masked by external elements (e.g. the state of the coin the agent flips may randomly match the state of one of the other coins before the flip action, and the agent incorrectly concludes that the state of the other coin is an important condition for the flip action).

Each of these issues can be tackled, to some extent, by performing statistical significance testing, and the planning operator learning mechanism presented in chapter 6 is based on this technique.

Standard dynamic programming techniques can build a utility map of a state space by cycling through each state, taking the best available action (according to the current estimate, or a random action in order to explore) and, when a reward is encountered in the following state, feeding this reward back to the previous state. The number or values which must be calculated is, however, exponential to the number of features present in the state space. This is referred to as the "curse of dimensionality" [73]. Attaching utilities to the operators removes the need for storage of these values, but poses a new set of challenges:

- Each planning operator's conditions represent only a small proportion of all the possible conditions of each state. The utility estimate attached to the operator is therefore an *aggregation* of many states from the full state-space.
- Planning operators are applied in parallel to calculate the following state. The agent therefore needs a mechanism for deciding the contribution made by each operator to the utility of taking a particular action.
- Total utility in a reinforcement learning system increases (or decreases) as the learning progresses. Operators with fewer conditions will increase (or decrease) in utility as a consequence of being applied more regularly, while those with more conditions will learn more slowly.

The general framework of utility-based action-selection is provided by dynamic programming (for model-based approaches), and reinforcement learning (for model-free approaches) [87]. The approach used in this research is initially model-free, learning the model from experience and can therefore be seen as fitting into both fields. The utilities learned by the agent provide estimates of the utility of being in a particular state and the approach therefore fits into the field of approximate dynamic programming (ADP) [73].

A range of model-based learning techniques have been proposed for agent-based planning mechanisms. The work presented here builds on contributions from several sources:

- Model based reinforcement learning: Dyna-Q [88].
- Planning operator learning: multi-stream dependency detection [64], noisy deictic rules [67] and association rule mining [1].
- Factored state mode approaches for decision theoretic planning [10].
- Approximate Dynamic Programming [73].

1.1 Motivation

The hypothesis of this thesis is that utility estimates attached to acquired parallel stochastic planning operators, describing the dynamics of a predictably probabilistic environment, can be used to compactly model the effectiveness of taking actions in that environment.

The general motivation for the work is to create agents that are both autonomous learners and who's behaviour is comprehensible by human designers. The drive comes largely from the author's commercial background in computer game agent programming. Games companies are generally reticent to use black-box techniques (such as a neural network), despite their obvious ability to deliver complex AI with reduced designer input, because a bug found in a solution requires a complete re-train. This newly re-trained solution can itself contain errors, and the risks are perceived as too great when it is considered that the error may only be discovered a week away from shipping a title with a multi-million dollar budget [12].

The use of rule-based models allows designers to either re-write rules by hand or, alternatively, interpret the errors by investigation and make adjustments to parameters or learning conditions when generating new rules.

Attaching values to rules means that the policy itself can be interpreted by designers, because they can see which actions and rules are favoured by the system in certain situations.

The particular properties of many computer game agent environments that make this technology applicable are that:

- An accurate model of the dynamics of the environment, from the perspective of an individual agent, is not known in advance, and often cannot be created due to the stochastic nature of the environment or the unpredictable actions of agents within it.
- Experience can be gathered through trial runs with negligible cost, as opposed to the cost of, for example, robot trials in potentially hazardous environments.

Some of the properties of the system that provide an advantage as a computer game agent controller include:

- Intelligible rules: the system creates rules that can be read and understood by a human designer.
- The rules can be modified by hand if necessary.
- The system can generalise over unseen states and therefore produce intelligent behaviour based on knowledge gained in similar situations.
- Limited processing power is required at run-time, with the learning occurring off-line.
- The design of AI agent controllers for computer games is an expensive process, often requiring highly skilled and experienced developers with extensive domain knowledge of each game. Automating this process could lead to significant cost savings and improvements in computer games.

1.2 Aims & Objectives

Techniques exist for creating effective agent controllers which exhibit some of the properties outlined above, but not all. The overall aim is to produce an effective and practical technology that inherits aspects of the best of these systems and exhibits each of the above desirable properties.

- *Create a framework for environment modelling agents:* the framework should be adaptable, in that a variety of environment modelling systems and action selection mechanisms can be incorporated.
- *Design a rule-based environment modelling system:* the system should have the expressive power to model the environment from the point of view of the agent. It must, therefore be able to model events that happen outside the agents control (environment actions), unpredictable/stochastic action outcomes, outcomes that are both independent and non-independent, and combinations of these. The rule system also needs to be in a human readable form and preferably in a form familiar to AI researchers in order to enable "glass-box" interpretation.
- *Design a system for learning the rule-based environment modelling system from experience:* human designers are not adept at creating probabilistic rule systems by hand. The environment modelling system should be able to acquire a model by analysing the environments response to action. The system should build a set of

operators by discovering patterns in changes to the environment in response to actions (or when no action is taken).

- *Design a system for attaching utility estimates to the rules:* allowing compact storage and human interpretable values to be attached to rules. The system should have the capacity to build rule utility estimates from successor rule utility estimates, without the need to enumerate the value of every state, or state-action pair in the environment.

1.3 Framework

The framework for the agent's learning consists of the following elements:

1) *Embodiment:* the agent is embodied, and situated in an environment: it can select actions (behaviours) available through its body and receives percepts, which are a function of the current environment state.

2) *Modelling:* the agent builds a model of its perception of the environment using parallel stochastic planning operators (P-SPOs). These are learnt empirically by observing the effects of actions through percepts. The percepts before and after each action are used as training data for a P-SPO learning algorithm. Note that the environment itself may be deterministic, but viewed through the agent's percepts, can appear stochastic.

3) *Policy generation:* the agent builds a policy by simulating actions using the environment model encapsulated in the rules. In initial tests, the agent uses standard dynamic programming to build a policy from simulated experience extracted from the model. In the full system, value estimates are attached to each operator. The operators contain actions and are therefore acting as a set of aggregation estimates encapsulating information in the form: taking action, a, under conditions, c, has utility, u.

A useful property of the framework is that the policy generation phase is entirely simulated and can therefore be seen as "free" in terms of cost to the agent in the environment. Additionally, the agent's goals can be changed, but the rules describing the environment's dynamics remain unchanged. It can therefore be set new tasks or goal without the need to re-model the environment [90].

1.4 Contributions

The research makes contributions in the following areas:

- *A framework for developmental AI is created:* a world model learning phase is followed by a planning phase using approximate dynamic programming. Extensions for in-line learning are explored.
- *Parallel Stochastic Planning Operators (P-SPOs) are defined:* an extension of Noisy Deictic Rules [67] to include provision for independent outcomes.
- *Apriori Stochastic Dependency Detection (ASDD) is defined & evaluated:* a fast stochastic rule learning algorithm for construction of P-SPOs from observation data using statistical significance and data mining methods.
- *Rule Value Reinforcement Learning (RVRL) is defined & evaluated:* a state aggregation method for approximate dynamic programming, using P-SPOs as aggregation estimates for a state aggregation function.

Experimentation is performed to evaluate:

- The performance of P-SPOs as an environment model for a dynamic programming based policy generator.
- The performance of RVRL in generating policies for agent action.

1.5 Structure

Chapters 2 and 3 provide background for environment modelling techniques from the perspective of agents, and methods of planning (policy formation) using the model. Chapter 4 provides the overall model-based learning framework used in the research. Chapter 5 defines parallel stochastic planning operators (P-SPOs). Chapter 6 defines the ASDD algorithm and associated functions for learning P-SPOs from data. Chapter 7 defines the test environments used in this research. Chapter 8 provides the results of the ASDD rule learning algorithm in terms of environment modelling and policy generation using a standard dynamic programming algorithm. Chapter 9 then defines the RVRL algorithm for attaching rule values to operators. Chapter 10 shows the results of the complete system, with values attached to operators learned in the framework and used as a policy. Chapter 11 discusses the achievements of the system, related work and future improvements to the system.

1.6 Previous Publications

The framework presented in chapter 4 and evaluation technique (chapter 8) have previously been presented and evaluated in conjunction with a multi-stream dependency detection based

technique for planning operator learning in *"SMART (Stochastic Model Acquisition with ReinforcemenT) Learning Agents: A Preliminary Report."* [17].

The ASDD method for learning stochastic logic rules (chapter 6) was defined and evaluated in *"The Apriori Stochastic Dependency Detection (ASDD) Algorithm for Learning Stochastic Logic Rules." [18]*.

Rule Value Reinforcement Learning (RVRL) for attaching values to planning operators (chapter 9) was first presented in *"Rule Value Reinforcement Learning for Cognitive Agents"* [16] and further evaluation in the context of an embodied agent environment modelling framework was published in *"Learning to Act with RVRL Agents"* [15]. Peer review comments from this and extensions to widen the applicability of the system have been incorporated in the approximate dynamic programming based update functions for RVRL presented in this work.

2. Background I: Agents, Environments & Models

This chapter introduces the agent and environment definitions which underpin this work. Embodied agents, with environment interaction mediated through action selection in an agent body, are defined and presented in the context of both deterministic and stochastic environments. Techniques for representing an environment model from the perspective of an agent are presented. The model representations are chosen because they define the evolution of the environment in response to agent action (allowing planning) and can be can be acquired from data.

2.1 Agents

The purpose of the system presented in this work is to create effective controllers for autonomous agents. A broad definition of an agent, as given by Wooldridge [98], is:

"An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives".

This definition puts no requirements on the agent to be part of a multi-agent system, to be able to communicate, or any of the other uses for which agents are employed. It simply defines an agent as a decision maker, situated in an environment.

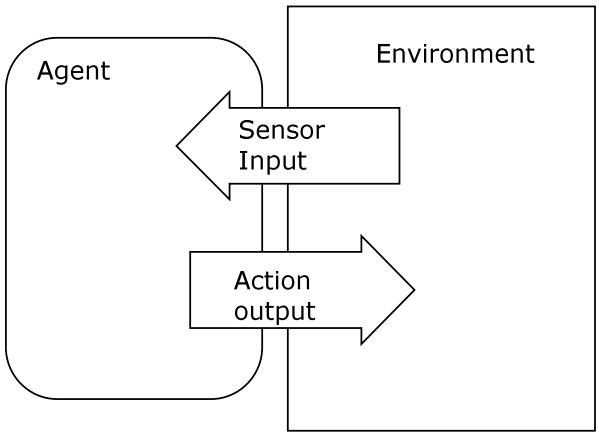


Figure 2.1: An agent and its environment. The agent produces actions in response to sensory input.

Figure 2.1 shows that the agent responds to sensor input from the environment with actions. This definition is broad in that there is no requirement for the agent to respond with intelligent decisions, and the type of environment is not defined. The agent is situated in an environment, but is also separate from it in that it's decision making process is outside the environment.

If the agent is treated as a separate decision making entity, outside the environment, then flexible agent architectures can be produced (as investigated by the EU SOCS project [85]). A good analogy is to consider a human playing a computer game. The human has a view of the world and can select actions through the controller, but these actions do not directly change the environment. Instead, they are stored as the *next action* that will be taken by the player when the game-world updates. The human player can easily be replaced by an artificial intelligence. If we give the AI a view of the world and allow it to trigger the same actions, then it should require no further change to the game world to integrate the AI. This view of embodiment is explored further in section 2.1.4.

The following section gives an abstract definition of agents and environments. The term agent is, in general, somewhat loosely defined and has been used in the definition of complex environments and interactions. Rather than debate these points, the intention here is to provide a useful definition of agency based around an embodied "perceive, deliberate, execute" framework, which will enable definitions from dynamic programming to be set in an agent context.

2.1.1 Agent: Action Selection within an Environment

A useful starting point for defining agents and environments is given by Wooldridge [98]. An environment is assumed to have a set of possible world states S, where $S = \{s_1, s_2, s_3, \ldots, s_n\}$. At any given time the environment can be in one of these states. Environments can have an infinite or a discrete number of states. An agent has a set of actions, A, which can influence this state, where $A = \{a_1, a_2, a_3, \ldots, a_n\}$.

The agent's purpose is to choose an action (make a decision). An agent can be viewed as a function, mapping a history of environment states, S^*, to an action.

$$action : S^* \rightarrow A \tag{2.1}$$

A *reactive* agent is an agent with no memory, which can only take account of the current state of the environment when deciding upon its next action. It is, therefore, defined by the function:

$$action : S \rightarrow A \tag{2.2}$$

A deterministic environment can be modelled as a function mapping the current state of the environment and the agent's action to a new state:

$$env : S \times A \rightarrow S \tag{2.3}$$

Starting from a state $s \in S$, the execution of the environment function on an action $a \in A$ produces a new state. A non-deterministic environment can be modelled as a mapping from state and action to a set of next states:

$$env : S \times A \to \wp(S) \tag{2.4}$$

This model appears simple but succinctly defines the agent as being a decision making entity, separate from its environment, but able to influence the environment. The environment function can be as complex as is required, containing multiple agents or just one single agent.

This abstract definition is simple and can be used to describe almost any agent, but it does not help in the practical construction of an agent.

Wooldridge provides a next step, which is to add perception to the agent, which captures the agent's ability to sense its environment, and that this sensing is an incomplete representation of the environment state. The see function takes a state and outputs a percept, where P = $\{p_1, p_2, p_3, \ldots, p_n\}$.

$$see : S \to P \tag{2.5}$$

The *action* function is then altered to become a function of the history of percepts for a cognitive agent, or a single percept for a reactive agent:

$$action : P^* \to A \tag{2.6}$$

Note that the mapping from states to percepts is generally not one-to-one. The number of possible percepts is less than or equal to the number of possible states, with several different states may mapping to the same percept. From the agent's perspective, two states that map to the same percept are indistinguishable. If each state maps to exactly one percept, then the environment is said to be fully observable.

The agent architecture employed in this work uses this form, with the additional layer of separation provided by the agent body, which is part of the environment and is the agent's only method of interaction with the environment (see section 2.1.4).

2.1.2 Agent: Perceive, Deliberate and Execute

A second perspective on the basic agent architecture is given by Ferber [30]. The definition, again, separates the agent's decision making process from the environment, but is more explicit in including multiple agents making simultaneous decisions in the environment. The agent is considered as three functions:

- Perceive

- Deliberate
- Execute.

The *perceive* function is defined separately for each the agent in the system. It associates a percept with each state of the world and can be defined as a mapping from the state of the environment to a set of available percepts for an agent. An agent, *g,* has a perceive function defined as:

$$perceive_g : S \rightarrow P_g \tag{2.7}$$

The *deliberate* function for a reactive agent is equivalent to that defined by Wooldridge. It takes a history of percepts and produces an action. Reactive agents have no memory or state and therefore the *deliberate* function consists simply of a reaction, modelled here by mapping a percept directly to action:

$$deliberate_g : P_g \rightarrow A \tag{2.8}$$

Cognitive agents have the ability to retain information, and thus act on the basis of perceptions and past experiences. Their deliberation process is therefore divided into memory and decision functions. The agent's capacity for memory can be characterised by an internal state $s_g \in S_g$ (the set of internal states of agent *g*). Memorising an experience is defined as moving from one internal state to another. The memorisation function takes an internal state and a percept and produces a new internal state:

$$mem_g : P_g \times S_g \rightarrow S_g \tag{2.9}$$

The deliberate function of a cognitive agent takes a percept and an internal state and produces an action to perform:

$$deliberate_g : P_g \times S_g \rightarrow A_g \tag{2.10}$$

The *"execute"* function in Ferber's definition, takes an individual agent's action and the current environment state and produces a set of influences, which will be combined by the environment function to transform the world's state.

$$execute_g : A_g \times S \rightarrow \Gamma \tag{2.11}$$

These influences are intended to resolve issues with ordering of execution in the agent environment. If execution order is not an issue, then the execute function can be adapted to directly transform the environment state:

$$execute_g : A_g \times S \rightarrow S \tag{2.12}$$

The environment function (action of the environment) can be represented as a special type of agent producing influences:

$$\begin{aligned} &environment : S \rightarrow A_E \\ &execute_e : S \times A_E \rightarrow \Gamma \end{aligned} \tag{2.13}$$

If execution order is not an issue, then the environment function can be mapped as a direct translation from state to state:

$$environment : S \rightarrow S \tag{2.14}$$

2.1.3 Environment Update Function: Direct Action, Discrete Time

An environment update function using the direct execution of actions for multiple agents (adapted from [30]) is presented below. The algorithm updates the state by processing the action of all agents in the environment, and then updates the state using the environment function. The environment function in this case could be replaced by a function $execute_e$, but keeping the function explicit will aid explanation in the following sections.

```
directEnvironmentUpdate(S,A,E)
   for all (a∈A) {
      p = perceive_a(S);
      a = deliberate_a(p);
      S = execute_a(S,a);
   }
   S = environment(S,E);
```

Algorithm 2-1: directEnvironmentUpdate. S = state, A = agent, E = environment definition

A variation on this algorithm forms the update function at the core of almost all current computer games (see [59] and [38] for examples). Computer games use discrete time updates so that a predictable performance occurs each time the algorithms are executed (avoiding differences due to precision errors). These discrete time-steps can be set to varying amounts for different aspects of the update, such that the agent's decision process may execute in 100 millisecond steps, while fast moving objects are simulated every 10 milliseconds.

For computer game simulations, the agent decision making processes is required to be flexible, such that a human could take the place of the agent as decision maker, with minimal code changes. This can be achieved using the embodied agent concept (below), in that the agent's execute function has the effect of changing the agent's body state, so that a new action of behaviour is selected, which is executed as part of the environment update.

2.1.4 Embodied agents

The concept of an agent body is often useful in drawing the boundary between agent and environment in both robot control [11] and reinforcement learning. Sutton and Barto make the following observation [87]:

> *"In particular, the boundary between agent and environment is not often the same as the physical boundary of a robot's or animal's body. Usually, the boundary is drawn closer to the agent than that. For example the motors and mechanical linkages of a robot and its sensing hardware should usually be considered parts of the environment rather than parts of the agent…*
>
> *Anything which cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of the environment."*

The agent perceives the world through the sensors of this body and acts in the world by triggering actions (behaviours) in the body. The agent itself is therefore detached from the environment in two ways:

- *The agent's actions do not have a direct effect on the environment:* the effects of an agent body on the environment mediate them.
- *The agent's perception of its environment is not a complete picture of the environment:* it is a reflection of the state of that environment as filtered through the agent body's sensors.

Embodied agents make explicit the distinction between the agent, its interface with the environment, and the environment itself. The agent's only means of gaining input from the environment is through sensors, which the environment updates. The perceive function maps these sensors to a percept. The agent's only means of output to the environment is by selecting actions. The selected action is stored in the body state. These actions are then executed by the agent body through its update function. The agent body is, in this respect, no different from any other object in the environment.

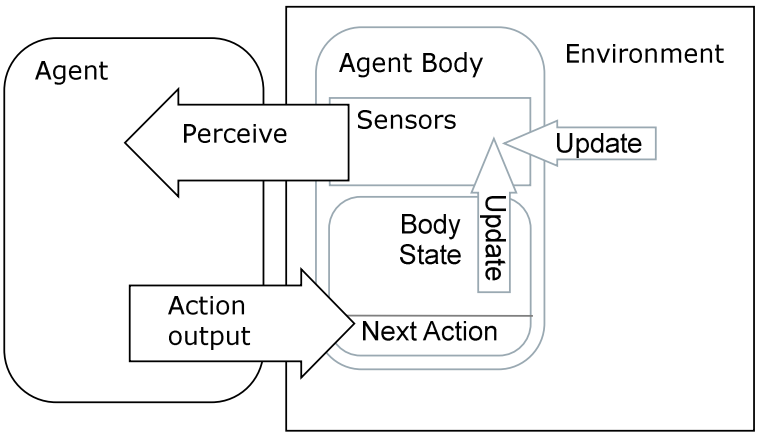


Figure 2.2: Embodied agents. The agent is a separate decision making entity whose contact with the environment is mitigated through an agent body. The agent selects the next action to be executed by the body and receives input by converting sensor information into percepts. Sensors gather information from the environment (including the agent's body).

The agent itself can be thought of as the mind of the body. If the required interface exists between the agent and its body, the mind could be considered to be operating outside the environment. The environment can proceed without intervention from the agent, with the environment acting as an external control mechanism. The agent body would, of course, be inactive without the agent's selection of actions, but its state can still be changed by the environment.

At a high level, the logical representation of the agent is unchanged from that presented in section 2.1.1, because the agent's body is part of the environment, and therefore remains part of the environment function. It is now possible, however, to be more explicit in modelling the environment as a set of objects with state, and the agent body as a special object with a set of available actions and a sensing mechanism.

The environment function below takes as input the current state, S, the set of agents, A, and the environment definition, E.

```
environment(S,A,E)
   for all (o∈S) {
      S = update_o(S);
   }
   S = update_E(S);
   for all (a∈A) {
      a.body.sensors=sense_a(S);
   }
```

Algorithm 2-2: environment. S = state, A = agent, E = environment definition.

The environment (including the agent body) is modelled as a set of objects, which are updated each time the environment updates. If the environment updates by a uniform amount each time, this is referred to as a discrete time environment.

$$update_o : S \to S$$
$$update_E : S \to S \tag{2.15}$$

Initially, all objects update the state of the environment. Next, the environment function updates all objects. Finally, all agent sensors are updated by mapping the current state of all environment objects (including the agent body) to the sensors using each agent's *sense* function.

The body based update function can now be defined. The update for all agents is altered to reflect the fact that the agent can only update its percept based on the contents of its sensors, and can only influence the environment through the changes to its body state. The execution of actions is no longer part of the agent, and is instead part of the environment update function (defined above). The individual agent's action changes the agent's body state by selecting actions.

```
bodyEnvironmentUpdate(S,A,E)
   for all (a∈A) {
      p = perceive_a(a.body.sensors);
      a = deliberate_a(p);
      a.body.state = execute_a(a.body.state,a);
   }
   S = environment(S,A,E);
```

Algorithm 2-3: bodyEnvironmentUpdate. S = environment, A = agent, E = environment definition.

The agent uses it's perceive function to receive a percept from the sensors and sends a choice of action to the agent body each time the environment is updated. The effects of the agent's actions are mediated through the agent body and its interactions with other objects in the environment.

Note that this is an approximate architecture, aimed at showing the interface between the agent and its body. Simulated environments such as those used in computer games can add multiple levels of complexity to incorporate factors such as collision detection, event propagation and network capabilities. The aim here is to provide the simplest environment definition which demonstrates the separation between the agent and its environment, while being flexible enough to incorporate more complex architectures.

This distinction between agent body and environment is broadly similar to that defined by the PROSOCS agent template [86]. The architecture is flexible in that it can be applied equally to agents embodied in robots, virtual agents (e.g. NPCs) in computer game environments embodied in avatars, or to any system in which a *view* and *controller* separates the agent from its environment.

Although more sophisticated formal definitions of agency exist, these are mainly geared toward refinements for individual agent types. The representation used is complete for the purposes of this research and is compatible with that used by Markov models which form a key element of the framework presented in chapter 4.

2.2 Environment Model

The term environment model is used to describe a function which can provide simulated experience of an environment. For a fully observable deterministic environment, this is a function mapping states and actions to successor states.

An environment model, from the perspective of an agent, can refer to any system that the agent can use to predict the outcome of its actions. Given an input state, *s*, and action, *a*, a model gives a prediction of the successor state, s'. For a deterministic environment this will take the same form as a sample model:

$$sampleModel : s \times a \rightarrow s' \tag{2.16}$$

If the environment is stochastic (random) then each state and action can lead to a set of possible next states, *S'*, with each member of the set having an associated probability, *Pr*. A *distribution model* is a model that produces all possible successor states and associated probabilities:

$$distributionModel : s \times a \rightarrow \{\{s_1, pr_1\}, \{s_2, pr_2\}, \ldots, \{s_n, pr_n\}\} where \sum_{n}^{1} pr_n = 1.0 \qquad (2.17)$$

A *sample* model is one that produces a single successor state and reward combination, sampled according to the probability of the successor state occurring. A distribution model of a coin flip would produce the output set: $\{\{heads, 0.5\}, \{tails, 0.5\}\}$.

A *sample* model for the same situation would produce one of the outputs at random. For example: $\{heads\}$.

An accurate *sample* model is one that produces a perfect simulation of the experience an agent would gather if it were to actually take a particular action in a particular state. An accurate distribution model generates all possible experiences that the agent would gather if it took a particular action in a particular state with a correct probability associated with each.

2.2.1 Markov Models

The environment model presented above can equivalently be described in the language of Markov models. Markov models are used to model *stochastic dynamic systems*. These are systems that are in one of a distinct number of states at a particular time, and which change states in response to *events*. Events can be outside the agent's control, known as environment events or exogenous events [7], or can be under the agent's control (its actions).

A Markov model defines a set of probability distributions describing the transition between a current state and next state depending on the past states. The system evolves in stages, where each event produces a transition from the state at time *t-1* to the state at time *t*.

In general, a discrete-time, stochastic dynamic system models the probability that the system with be in a particular state at time *t* given a history of previous states:

$$\Pr(S^t \mid S^0, S^1, \ldots, S^{t-1}) \qquad (2.18)$$

A common simplifying assumption, applicable to a large range of systems, is that the current state contains enough information to predict the next state. This assumption is known as the *Markov assumption*. If this assumption holds, the history of states becomes irrelevant to the prediction of the future state:

$$\Pr(S^t \mid S^0, S^1, \ldots, S^{t-1}) = \Pr(S^t \mid S^{t-1}) \qquad (2.19)$$

If the effects of the event are independent of the time at which the event occurred, and depend only on the current state of the environment, then the model is said to be stationary. The models presented in this research assume that the state transitions can be modelled using the

Markov assumption, and are stationary, finite-state, discrete-time, stochastic dynamic systems.

Perceptions

If we assume an implicit event model (where exogenous events are modelled as part of the agent action) and we assume observations are independent of time. The probability of receiving a particular percept at time t can depend on:

- The state of the system at time t-1,
- The action taken by the agent at time t-1.

This model can be used to describe a variety of assumptions about the sensing capabilities of the agent. Each of these assumptions corresponds to a type of Markov decision process.

- *Fully observable* MDP (FOMPD): The agent's observations exactly describe the state that it is in at time t. The agent, in effect observes the complete state of its environment (P = S). The agent therefore receives perfect feedback about the results of its actions and the effects of *exogenous* events (events outside its control).
- *Non-observable* MDP (NOMDP): The agent receives no perceptual information from its sensors. This can be modelled as P = {p}, indicating that the agent receives the same percept at all times or as P = {} (indication that the agent receives no useful perceptual information).
- *Partially observable* MDP (POMDP): The agent receives incomplete or noisy information about the state of its environment.

This research treats all environments as FOMDP, despite the fact that we are making incomplete observations on the state of the environment. Environments are treated as FOMDP because the agent does not have any world model other than that it can gather from evidence. Therefore the problem is fully observable in terms of the possible models that the agent can construct. The world to be modelled by the agent could be inherently stochastic or, in fact be deterministic, but adhering to laws of which the agent has no knowledge and must therefore approximate by use of probabilistic rules. For a full discussion of these distinctions see Pearl [69].

2.2.2 Perceptual Model

The separation of the agent from its environment provides a useful abstraction for agent modelling because the agent's environment is often too complex to model completely.

The sole contact an agent has with its environment is through the percepts received and actions selected. The agents in this research must make a model of their environment through interaction. This model must be based on the knowledge of the actions it has selected and the percepts that it has received. The agent's task is not, therefore, to model the environment itself, but rather to model its perception of the environment.

A perceptual model is logically equivalent to an environment model from the point of view of the agent. It models the successor percept it will receive, given an initial percept or action. Given an input percept, p, and action, a, the perceptual model gives a prediction of the successor percept, p'. If the function is deterministic or a sample mode is used this will be of the form:

$$samplePerceptualModel : p \times a \rightarrow p' \tag{2.20}$$

The perceptual model can be deterministic even if the environment itself is stochastic. The mapping of states to percepts may make the elements of the environment which contain stochastic properties map to the same percepts. For example, a tic-tac-toe agent controlling a robot may have a percept that is a function of the current board state. Taking an action would predictably change the percept, but the state of the world outside the percept (including whether its opponent had decided to stop playing) could change randomly.

As is the case with the state-based mode, if the environment is stochastic from the perspective of the agent, then each percept and action can lead to a set of possible next percepts, P', with each member of the set having an associated probability, Pr:

$$\begin{aligned} &distributionPerceptualModel : p \times a \rightarrow \{\{p_1, pr_1\}, \{p_2, pr_2\}, \ldots, \{p_n, pr_n\}\} \\ &where \sum_{n}^{1} pr_n = 1.0 \end{aligned} \tag{2.21}$$

A sample model chooses a single percept from P', distributed according to the associated probability.

The environment may be deterministic, but appear stochastic from the point of view of the agent, because its mapping is not one to one. The use of a probability based approach enables the agent to model complex environments at a high level of abstraction. It also enables it to model events which occur randomly, and model events which are not random, but which appear random from the limited perspective of the agent.

For each action, a_t, taken at time t, after receiving a history of percepts $\{p_1, \ldots, p_t\}$ an accurate model predicts the percept, p_{t+1}. A perceptual model exhibits the Markov property if all the information required to predict percept p_{t+1} is present in p_t. Often the Markov property will

not hold for the agent's perception of its environment, but the agent can make an approximate model of the environment using the Markov assumption.

2.3 Model Representation

A model of an environment is a representation, or abstraction, which captures essential details of the environment. For an agent, these essential details should contain everything it needs to know in order to make decisions to move the environment towards its preferred state (goals). For an agent to be able to make a *plan*, it must be able to model its environment in order to predict the result of each action it takes.

Models represent the expected transition for a current state to a successor state, in response to an action. A simple *tabular method* for representing this transition, presented in Dyna-Q [88], contains a table entry for each state action pair, followed by all possible successor states and their associated probabilities. This tabular method can be referred to as a "state-action" map. For a deterministic environment, each table entry contains a single transition, $s_t, a_t \rightarrow s_{t+1}$. If the environment is stochastic, then there may be several following states with a probability of reaching each one.

A simple example will help to illustrate these concepts. Consider an agent in an environment containing a coin, showing either heads or tails. The agent has two possible actions: *flip* or *do nothing*. If the agent chooses the *flip* action, then the result will be heads 50% of the time (probability 0.5) and tails otherwise. If it chooses to *do nothing* the coin will remain as it was.

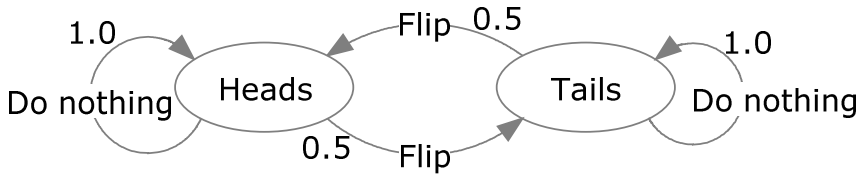


Figure 2.3: States transition diagram for a coin flipping agent. States are represented by ovals and actions by arrows. Arrows lead from the start state to the end state for a particular action labelled with a probability.

Figure 2.3 shows a state transition diagram for the coin flipping agent. Nodes correspond to states. Arcs show possible transitions between states in response to actions. Each arc is labelled with the action and associated probability of the state transition.

Table 2-1 gives an example of a tabular world model representation of this environment.

Table 2-1: A tabular world model built by labelling states using empirical evidence

State	Action	Next State	Prob.
Heads	Do Nothing	Heads	1.0
Heads	Flip	Heads	0.5
		Tails	0.5
Tails	Do Nothing	Tails	1.0
Tails	Flip	Heads	0.5
		Tails	0.5

The *do nothing* action is an example of a *deterministic* action: an action that always has the same result. The *flip coin* action is an example of a *stochastic* action because it can have more than one outcome, each with a probability of occurring.

The agent could equivalently model its perception of the environment using table entries for the transitions $p_t, a_t \rightarrow p_{t+1}$. If the environment is fully observable, then the state transition diagram and tabular world model are identical for both state models and perceptual models.

2.3.1 Factored State Models

The *states* of the environment can be described by a number of *features*. The model presented above has only two states, and the only feature of those states is the *coin side*. Further features could be added to the model to describe, for example, the current wind speed. Alternatively each state can be given a separate label. Table 2-2 shows these two equivalent representations.

Table 2-2: Coin Side and Wind Speed are features in a factored state model. The state of the world can be described by the states of each of the features that describe it, or by a label defining the combined states.

Coin Side	Wind Speed	Labelled States
Heads	Strong	COIN_HEADS_WIND_STRONG
Heads	Weak	COIN_HEADS_WIND_WEAK
Tails	Strong	COIN_TAILS_WIND_STRONG
Tails	Weak	COIN_TAILS_WIND_WEAK

Factored state models are useful for agent planning systems for the following reasons (adapted from Sanner [79]):

- A world model can be described compactly using a factored state transition model.
- The agent's goals may be dependent upon only a small part of the world model. An incomplete model may describe the part of the environment influencing the agent's goal well, despite being a poor model of the complete environment.
- Structure in the model can be used to find structure in the utility function.

- The factored model can be learned more efficiently from data.

If, for example, the agent's goal is for the coin to be showing heads, then the wind speed may be found to be irrelevant to its preferred state. Assuming the agent has an accurate description of the affects of its actions on the world it can use a model with poorly evaluated transition probabilities for wind speed, but which is an accurate model for the agent's purposes.

Figure 2.4 shows how the factored state transition diagram for the coin, with the additional wind speed element, can compactly represent this additional feature. The diagram accurately represents the feature as being outside the agent's control and having no effect on the coin side state.

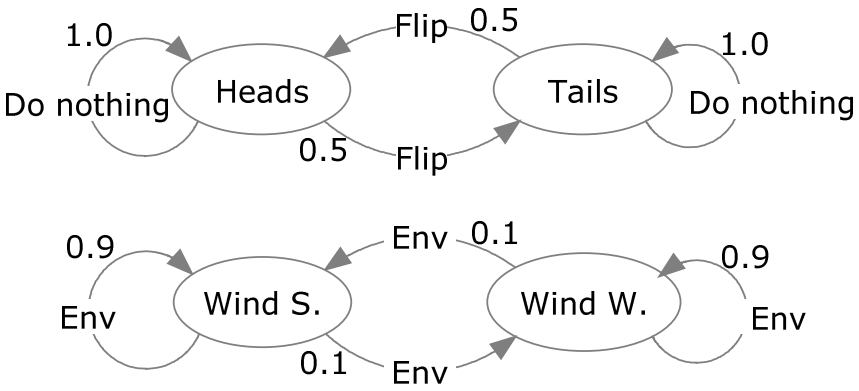


Figure 2.4: Factored state transition diagram of a coin flipping agent with an additional Boolean wind speed component that changes with probability 0.1 each time step, irrespective of the agent's action.

Figure 2.5 shows the equivalent state transition diagram for the coin flipping agent with the additional wind speed feature. This graph is quite complex, even with only two state features. The complexity is, in part, caused by the need for the arrows showing state transitions to incorporate the effects of environmental actions (transitions outside the agent's control).

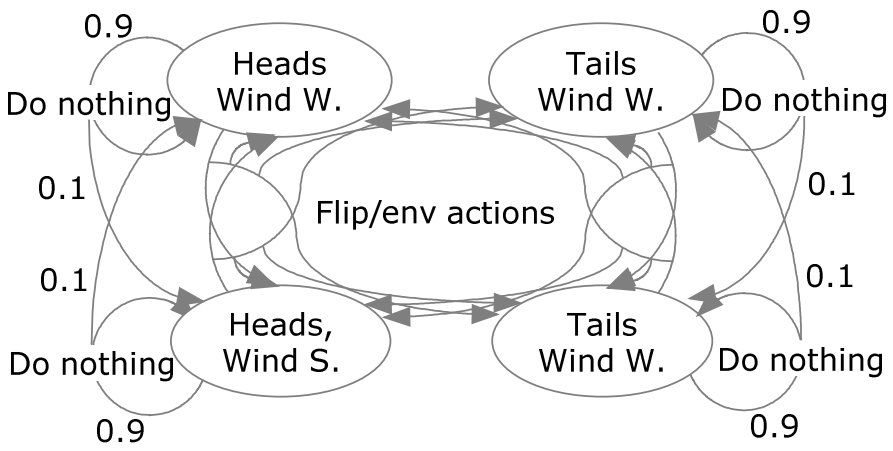


Figure 2.5: State transition diagram for the coin flipping agent with an additional wind speed feature. The probabilities for the "do nothing" action are shown. Flip action probabilities are omitted.

If the agent is attempting to model the effects of its actions, it can be useful to know the limits of each action's influence. This can also be useful when the agent is trying to plan. The agent can plan more effectively if its model takes into account the fact that some parts of the environment are beyond its control.

2.3.2 Influence Diagrams

Influence diagrams give a visual representation of a factored state model which clearly shows the dependencies between state variables. The representation is a graphical method with a similar structure to a Bayesian network [68]. If the environment model is stationary (does not change over time) and displays the Markov property (next state is dependent only on current state and action), then the model can be represented by a two-tier network, with a layer of nodes for the current state and a layer for the successor state, known as a temporal Bayesian network [7] or two-tier-Bayesian network (2TBN) [8].

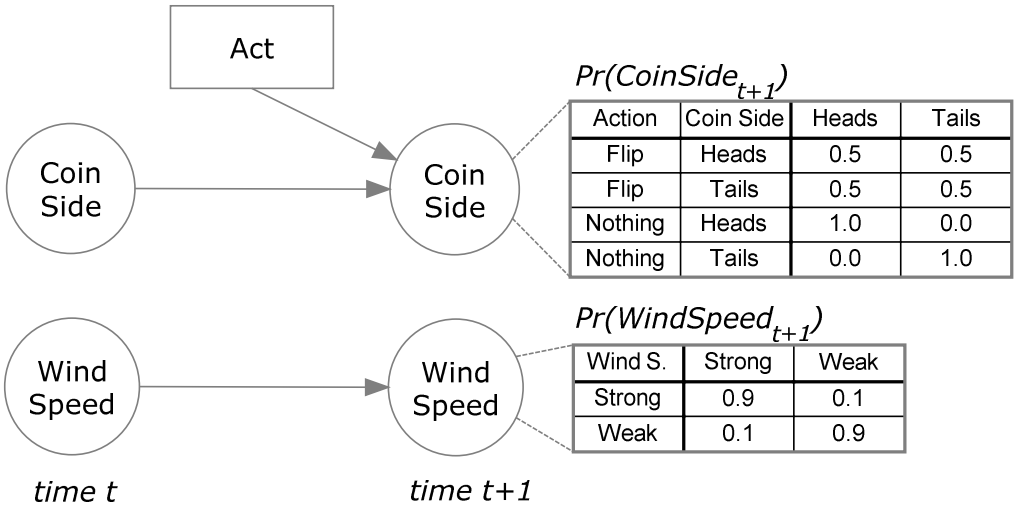


Figure 2.6: Influence Diagram representing a factored state model for a coin flipping agent.

Figure 2.6 shows an influence diagram representation of the coin flipping agent in factored state model form. Each output node has an associated conditional probability table, which shows the probability that each feature will take on a particular value, depending on the input. The diagram is a useful visualisation tool because it shows explicitly the connection between actions and state variables. There is also a reduction in storage in comparison to a non-factored representation. In this example the first table stores eight values and the 2nd four (totalling 12). A non-factored CPT would require 2 × 2 × 2 rows (combinations of action, coin side, wind speed) and 2 × 2 rows (coin side and wind speed outputs) totalling 32 entries.

Notice that the CPT displays some wasted storage. The flip action has entries depending upon the input being heads or tails. In fact this is irrelevant, and the CPT can be further simplified by using a tree representation for the conditional probability, known as a structured CPT [7]. Figure 2.7 shows this representation.

Pr(CoinSide$_{t+1}$)

Action	Coin Side	Heads	Tails
Flip	Heads	0.5	0.5
Flip	Tails	0.5	0.5
Nothing	Heads	1.0	0.0
Nothing	Tails	0.0	1.0

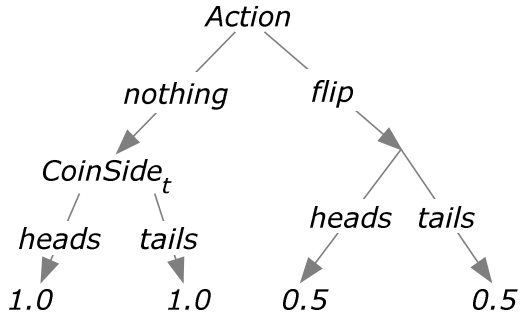


Figure 2.7: Structured CPT representation of conditional probability tables for influence diagrams.

Influence diagrams are an *explicit event* representation of two-tier-Bayesian networks (2TBN). Explicit event representations allow events to be represented as inputs to the network. Figure 2.6 shows an example of an influence diagram in which the agent's action is the only event input, represented as a special node in the network that is under the agent's control. The influence diagram representation allows for multiple events to be input, which would be represented as additional rectangular nodes. The events must be external to the features of the state space, because the model is not required to predict the occurrence of future events, and indeed the external input of agent control is necessary to the use of the model.

An alternative form of representation is to build a separate 2TBN for each action. A 2TBN is an *implicit event* representation and, therefore, does not contain input event nodes. Implicit event models include the influence of external events as changes to the transition probabilities in the network. The selection of an action requires the selection of a separate 2TBN model because there is no mechanism for actions (events) to be input. For a discussion of this representation, see Boutilier et al [7].

In this research, the agent's action is the only event type allowed in the model. The planning operator representation defined in chapter 5 allows additional event inputs but, because these are outside the agent's control, the model could not be used unless there was a way of predicting their occurrence. Instead, an implicit event model is preferred for all events except the agent action, with the model including the probability that external events will change the environment state in the transition function.

Closed vs. Open World Assumptions

The closed world assumption is defined in logic programming as the assumption that anything that cannot be shown to be true in the world is false [74]. The opposite of this is the open world assumption, which limits the deductions an agent can make to only those that it can show to be true or false (negation as failure is not used). If a deduction in the open world assumption cannot be shown to be true or false, then it has unknown value.

The systems presented in this work are closed, in the sense that everything that can be perceived in the world is present in the initial percept in one of the possible states, making negation as failure possible. The models are also open, in the sense that negation as failure is not used because all facts can be shown to be true or false at all times. The fixed percept size is a restrictive condition, but is useful in that it reduces the possible percept space to be static and non-infinite. The result is that there are no add/delete operators, only changes of perceptual features from one state to another. The planning operator system and operator

learning methods presented in chapters 5 and 6 are, however, easily adapted to learning add/delete operators, and this is an interesting area for future research (section 11.3).

2.3.3 Probabilistic STRIPS operators (PSOs)

Influence diagrams represent the stochastic evolution or persistence of each state variable in response to actions. Often, however, the agent's actions have influence on only a limited subset of the environment features. Figure 2.6 shows an example of an influence diagram in which one of the state variables, wind speed, is not dependent upon the agent's action.

In a wide variety of agent environments, there are a number of state variables that do not change unless the agent acts upon them, and it is advantageous to planning system to have a representation that models this persistence in an economical way. Influence diagrams require that each variable must be explicitly asserted as persisting in value if unaffected by an action. This is an example of the well known *frame problem* and is discussed in section 5.5.

One solution to this persistence problem, first developed for STRIPS operators [31], is to describe the outcomes of an action in terms of changes to the current state, leaving all unchanged variables unaffected.

A classical (deterministic) STRIPS operator has an *action*, a set of *preconditions* (`pre`), which define the situations in which the operator can be applied, and a set of *effects*, which describe the *change* to the environment if the action is taken. The change is defined in terms of *deletions* (`del`) and *additions* (`add`), to the current state of the world. An example operator which turns a coin from heads to tails is shown below:

```
operator turn(X)
  pre: showing(X, heads)
  del: showing(X, heads)
  add: showing(X, tails)
operator turn(X)
  pre: showing(X, tails)
  del: showing(X, tails)
  add: showing(X, heads)
```

Probabilistic STRIPS operators (PSOs) are a probabilistic extension to STRIPS introduced by Hanks [25][34]. The operators allow actions to be represented by multiple operators with different effects (such that an action can have different effects depending upon the context in which it is used) and include probabilistic effects (to model both stochastic actions and those for which the effects are not certain).

The conditions of operators in a PSO set are *mutually exclusive* (only one operator can be applied to a particular state) and *exhaustive* (each state has exactly one applicable operator). The context of each operator determines the conditions under which the (stochastic) effects of the operator will be applied. The effects are modelled as change sets, which define a set of additions and subtractions from the current state. Each member of the change set has an associated probability. The probability of the members of the change sets sums to one. Figure 2.8 shows a decision tree PSO representation of a *flip* action. Figure 2.9 shows a PSO representation of the *doNothing* action. Each action has a separate PSO. The actions can be represented by decisions trees because they are mutually exclusive. Each branch of the tree represents a *precondition*. The leaves of the tree contain additions (+) and subtractions (-) from the state, equivalent to the *deletions* and *additions* of STRIPS operators (abbreviated such that heads = H, tails= T, strong = S and weak = W).

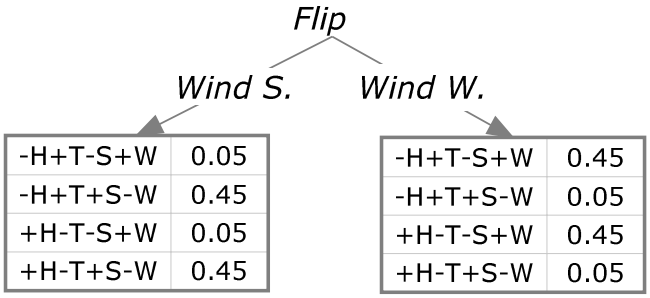


Figure 2.8: PSO representation of the Flip action.

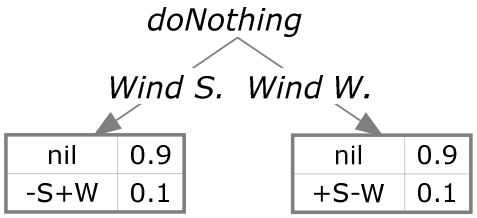


Figure 2.9: PSO representation of the doNothing action.

The example shows both the strength and the weakness of the PSO representation. The *doNothing* action is compactly represented, because the action does not change the state of the coin side, and its persistence needs no further representation by the action. The flip action, however, has to include each of the outcomes for wind speed in its change sets, despite the change being outside the influence of the operator. PSOs, much like STRIPS operators, can give a compact representation of an environment which changes solely in response to an agent's action, but become complex if the environment changes outside the agent's control.

2.3.4 Noisy Deictic Rules (NDRs)

Noisy deictic rules (NDRs) are a relational extension of PSOs which include deictic references and relax the frame assumption by including *noise* to model changes to the environment outside the agent's control. NDRs [67] are an extension of Probabilistic Relational Planning Rules [66][100]. NDR's are also known as noisy in-deterministic deictic rules, or NID rules [53][54].

Similarly to PSOs, NDRs require that the rule conditions are mutually exclusive, such that any given state-action pair is covered by at most one rule. An example of a set of NDRs representing the coin flipping agent is given below:

$$
\begin{aligned}
&flip(X) : showing(X, heads) \\
&\rightarrow \begin{cases} 0.5 : showing(X, tails), \neg showing(X, heads) \\ 0.4 : no\ change \\ 0.1 : noise \end{cases} \\
&flip(X) : showing(X, tails) \\
&\rightarrow \begin{cases} 0.5 : showing(X, heads), \neg showing(X, tails) \\ 0.4 : no\ change \\ 0.1 : noise \end{cases} \\
&default : \\
&\rightarrow \begin{cases} 0.9 : no\ change \\ 0.1 : noise \end{cases}
\end{aligned}
$$

The probabilistic outcome set is similar to that used by the PSO representation. There are three rules in the rule set above. Each rule contains:

- An action: e.g. *flip(X).*
- A precondition: e.g. *showing(X, heads).*
- An outcome set with associated probabilities (indicated by the arrow, $\rightarrow$, with the sum of the outcome probabilities being 1.0).

Actions contain parameters which can be matched to the environment state to form rules. This allows generalisation in that the rules can be applied to many objects as long as the conditions of the rule hold. In the rule set above, the parameter *X* of the *flip* action can be matched to any coin in the environment, with the choice between the first and second rule dependent upon whether the coin was previously showing heads or tails. Exactly one rule can match each possible state/action combination of the environment and the outcome set determines the changes to the current environment state in response to the action.

The frame assumption is relaxed, in that each rule can have a *noise* component, as well as a *no change* component. In the above example, the change of wind speed is handled by the noise component of the rules, simplifying the flip action. The *do nothing* action has no effect and does not need to be included in this representation, because it is handled by the default rule.

The outcome of the noise result is not modelled by NDRs. If the noise outcome occurs, then the state transitions to one of the other possible environment states with equal probability, or a probability defined by a simple distribution (e.g. proportional to the observed occurrences of the state in training data). In the case of the coin flip environment, this results in a poor model, but has been shown to be an effective method in environments for which the noise component need only introduce a random element to the agent's planning mechanism to simulate that its plan may fail with a defined probability [67].

Deictic references are used to model variables that are required as context to the action, but are not required as parameters. If, for example, we wanted to improve our model of the flip action to include information on whether or not the agent is wearing a glove (which makes the action less likely to succeed) while attempting the flip action, we can model this as follows:

$$\begin{aligned} &flip(X):\{Y:wearing(Y),glove(Y)\} \\ &\qquad showing(X,heads) \\ &\rightarrow \begin{cases} 0.2:showing(X,tails),\neg showing(X,heads) \\ 0.7:no\ change \\ 0.1:noise \end{cases} \end{aligned}$$

The parameterised action, *flip(X)*, and context, *showing(X, heads)*, remain unchanged, but an extra line of deictic reference, *{Y: wearing(Y),glove(Y)},* allows the action definition to be applied with greater contextual information. In general, the term deictic reference follows the terminology of Agre and Chapman [2] in adding the ability to refer to objects on which an action is being performed. They can be used, for example, to refer to an object which is under the one that is referred to in the parameter list of the action; or the object currently in-front of the agent.

The restriction that only a single rule can be applied in a given situation is useful in planning terms, but, as the difficulty with the wind-speed environment-variable shows, is restrictive in terms of the ability to model a wide range of situations compactly. Although it is possible to model the environment accurately by adding extra conditions to the rules to include new operators for each wind-speed, this would require a number of rules exponential to the variables which change outside the agent's control.

Chapter 5 defines a rule syntax which solves the issue of state features which change outside the agent's control by allowing the application of rules in parallel, and through the addition of an *environment* operator.

2.4 Summary

This chapter has presented the background for the planning operator based environment modelling system that will be presented in chapter 5. An initial definition of an agent and its environment was followed by the introduction of an environment model based on embodied agents. The agent was defined as a decision making entity, separate from an environment, but able to influence it. The agent's only method of control is to select actions (behaviours) in the agent body, which are then carried out in the environment update cycle.

With the agent's interaction with its environment established, it was possible to define an environment model in terms of simulated environmental response to action. If the agent's access to the environment state is mitigated through its perception, then the model is of the agent's future percepts.

Methods of representing the model were presented: a simple tabular representation with probabilistic extension; probabilistic graphical models; and probabilistic planning operator representations, including probabilistic STRIPS operators (PSOs) and noisy deictic rules (NDRs).

3. Background II: Model Learning & Planning

This chapter introduces the required background for the stochastic planning operator based model learning algorithm used in this research, and the background in planning for stochastic environments based on dynamic programming.

3.1 Model Learning

If it is not possible to provide the agent with an accurate world model, then the agent can be tasked to learn this information from interaction in the environment. This can be achieved in simple environments through observation of the probabilities of transitions between labelled states. In more complex environments it is necessary to learn a factored state model, or a probabilistic relational model.

The simple *tabular model* presented in section 2.3 can be learned by the Dyna-Q [88] method. Each state is labelled as it is encountered, and a map is built of the following state after each action.

After each experientially encountered transition, $s_t, a_t \rightarrow s_{t+1}$, the model records a table entry for s_t, a_t with the prediction that s_{t+1} will follow. If the model is queried with a previously encountered state-action pair, it returns the last successor state as the model prediction.

If the environment is stochastic, then there may be several following states with a probability of reaching each one. The model can be extended to incorporate this by recording each successor state encountered, with a count of the number of times it has been visited. The ratio of each encountered successor provides an empirical probability for the model.

The coin flipping agent presented previously is a simple environment with two states {*heads, tails*}. This form of model is relatively easy for an agent to build from empirical evidence. It builds a list of all the states it has observed and the actions it took in each state. It then records the state it observes subsequently.

The number of times the next state occurred for each state-action pair, divided by the total number of occurrences of the state action pair gives the empirical probability. Table 2-1 gives an example of an agent's representation of a world model built in this way:

Table 3-1: Building a tabular world model by labelling states using empirical evidence

State	Action	Next State	Obs.	Empirical Prob:
Heads	Do Nothing	Heads	2104	2104/2104 = 1.0
Heads	Flip	Heads	1024	1024/(1024+976) =.512
		Tails	976	976/(1024+976) = .488
Tails	Do Nothing	Tails	1978	1978/1978 = 1.0
Tails	Flip	Heads	995	995/(995+1002) = .498
		Tails	1002	1002/(995+1002) =.502

The agent can equivalently model its perception of the environment by recording table entries for the transitions $p_t, a_t \rightarrow p_{t+1}$.

If the environment is complex this method becomes impractical because the number of states that a world can be in is exponential to the number of factors involved. For example: adding the two-state variable *wind speed* to the model, increases the total number of states from 2 (*heads* or *tails*) to 4 (2 × 2). If a further variable is added, *weight of coin,* with states *heavy*, *medium* and *light* then the number of states increases from 4 to 4 × 3 = 12.

The situation is compounded by the addition of continuous variables (variables with infinite, rather than discrete values). For example, *weight of coin* may be represented in kg. It can have an infinite number of states and the total number of states is therefore infinite ($4 \times \infty = \infty$).

3.1.1 Learning Factored State Models

In order to model the environment using features, it is necessary to create a model of the features which are dependent upon each other. An agent, attempting to build a world model in complex environments, must be able to determine which features are important and which to disregard. For example, the agent can take a number of *flip coin* actions, and observe the following information before and after each action:

- agent action (*flip coin* or *do nothing*),
- coin side,
- wind speed,
- wind direction,
- coin weight.

Humans are adept at making rules such as:

- If the action is *do nothing*, the final coin will be heads if it was heads previously and tails if it was tails previously.
- If the action is *flip coin*, half the time the result will be heads and the other half the result will be tails.

Features such as coin weight and wind speed are discounted from our rules because we assume that they do not have a *significant* effect. For an effect to be significant it must have an observable impact on the outcome.

The process of building a factored state mode from empirical data requires the identification of conditions relevant to the probabilities of the outcomes. The task for the learning process is to find the minimal set of conditions which accurately capture the outcome probabilities for each operator. Finding an operator set with minimal conditions is important because:

(i) All relevant data to the probability of an individual outcome can be included in the rule.

(ii) Associating utilities with the rules requires that rules group together related areas of the state-action space.

Several methods exist for learning factored state models. Influence diagrams are an example of a Bayesian network, and can therefore be learned using methods for learning Bayesian network structure. This thesis focuses on a planning operator based representation, and Bayesian network learning is therefore beyond the scope of this research. The interested reader is referred to the Bayesian network learning algorithm developed by Friedman and Goldszmidt [32], which outputs a network from data and uses decision trees to represent conditional probability tables.

A noisy deictic rule (NDR) learning mechanism [67] has been developed which uses an adaptation of inductive logic programming [62] at its core. This method is not able to model parallel rules with multiple variables changing outside the agents control and is, therefore not directly considered background to this research. A summary of the method is provided in section 3.1.5.

Section 3.1.2 outlines the Multi-Stream Dependency Detection (MSDD) algorithm: a probabilistic STRIPS operator (PSO) learning method that has been developed using a combination of structured search and statistical significance.

Section 3.1.4 outlines the Apriori algorithm: a method for extracting probabilistic association rules from large databases. This is an active area of research in data mining, with the Apriori algorithm forming the basis of several methods.

A combination of the MSDD and Apriori methods form the core of the Apriori Stochastic Dependency Detection (ASDD) algorithm, defined in chapter 6.

3.1.2 Learning PSOs with the MSDD Algorithm

The Multi-Stream Dependency Detection algorithm (MSDD) is an algorithm developed by Oates and Cohen that has previously been employed to learn probabilistic STRIPS operators [65]. The algorithm requires that the percept is of a fixed size and that the possible values of each percept element and action are known in advance. MSDD is a batch algorithm and uses the history of perceptual data observed by the agent to form probabilistic STRIPS operators. Each item in the history contains the initial percept, action taken and successor percept.

The algorithm starts with a single operator matching all conditions (the most general operator possible) and performs a search from general to specific over the possible dependencies.

The function, *f,* evaluates the best node to expand next. A typical measure would be to find the node with the highest occurrence in the history, by counting the co-occurrence of the node's preconditions in the initial percept and effects in the successor percept. This requires a complete pass over the data set of perceptual data, *D.*

The coin flip agent environment has actions, A, and percepts, P, defined as:

$A = \{flip, doNothing\}$
$P = \{CoinSide, WindSpeed, Reward\}$
$P_{CS} = \{heads, tails\}$
$P_{WS} = \{strong, weak\}$
$P_{R} = \{pos, neg\}$

As an example of a PDI produced by the agent in this environment, the 67th element of the percept history, D, could be:

$P_{66} = \{tails, weak, neg\}$
$A_{66} = \{flip\}$
$P_{67} = \{heads, strong, pos\}$

A rule in MSDD is essentially a PSO of the form <*action, conditions*><**, effects*> *probability*>.

Each rule must contain all elements of the percept, but can contain a special wildcard element (*), which matches anything. Actions are included as one of the condition elements, with this element forced to be a wildcard in the effects. One of the rules for the *doNothing* action would be represented in MSDD as:

Conditions = <doNothing, *, weak, *>
Effects = <*, *, strong, *>
Probability = 0.1

The above rule can be written as:

```
<doNothing, *, weak, *><*, *, strong, *> Pr: 0.1
```

PSOs do not change any variable which is not mentioned in the effects. The rule above does not, therefore, change the condition *weak* for *wind speed* with probability 0.9 (the remaining probability).

```
MSDD (D, f, maxnodes)
1. expanded = 0
2. nodes  = ROOT-NODE()
3. while NOT-EMPTY(nodes) and expanded < maxnodes do
    a. remove from nodes the node n maximising f(D,n)
    b. EXPAND(n), adding its children to nodes
    c. increment expanded by the number of
       children generated in (b)
EXPAND (n)
1. for i from m down to 1 do
   a. if n.preconditions[i] ≠ '*' then
      return children
   b. for t ∈ possible values of n.preconditions[i] do
      i.   child = COPY-NODE(n)
      ii.  child.preconditions[i] = t
      iii. push child onto children
2. repeat (1) for the effects of n
3. return children
```

Algorithm 3-1: Multi-Stream Dependency Detection (MSDD). D = set of perpetual data items, f = an evaluation function, maxnodes = the maximum nodes that can be explored.

The algorithm does not specify which children should be generated before others, but does ensure that each dependency is explored only once. The final node list is output in general-to-specific order as a natural consequence of the algorithm.

The experiments comparing MSDD to the ASDD algorithm (chapter 7) make three additions to the above algorithm.

The first is in EXPAND (3.b) in which a check is made that the generated child matches at least one observation in the percept history, D, before adding it to children. This stage is equivalent to the "REMOVE_PRUNABLE" stage in the original MSDD algorithm [64].

For example, MSDD can generate the rule:

```
<doNothing, tails, *, pos><*, tails, *, *> Pr: n/k
```

In the coin flip environment, the agent cannot receive a positive reward when the coin is showing tails. A check against the data set will reveal that the generated rule has no matches and can be eliminated from the node list, along with its children (which will never be generated).

The second change is to preclude the generation of children of rules which have a probability of 1.0. In other words, rules with an output that is already predicted with certainty by the input. In this case, no more accurate prediction can be made and there is no need to generate further children.

The third change is that the effect part of the rule is allowed to have only one effect element. This change has been made to match rules generated by the ASDD algorithm, in exhibiting conditional independence. Combining individual effects can generate complete successor states. If this single outcome restriction was not included, a large number of rules can be generated by standard MSDD. The disadvantage of this additional restriction is that illegal states can be created when multiple rules are applied in parallel, such as one containing `<*, tails, *, pos>`. These rules are eliminated using constraints (section 5.4.5).

The *filter* process of MSDD removes specific rules with effects which are covered by more general rules.

3.1.3 Filter

The *filter* function (Oates and Cohen [65]) is an extension to the MSDD algorithm. It removes rules that are *subsumed* and *covered* by more general ones:

- *Subsumed:* a rule, $r_{1,}$ is subsumed by a rule, r_2, if the PDIs matched by r_1 are subset or equal to the PDIs matched by r_2. In other words, r_1 is a more specific version of r_2.
- *Covered:* a rule, $r_{1,}$ is covered by a rule, r_2, if r_1 is subsumed by r_2 and there is no statistically significant difference between the rules.

If a rule is covered by another rule, then the increased specificity of the conditions does not have a significant bearing on the rules outcome. The statistical test for non-independence is achieved by using the G statistic (Wickens [97]). See appendix section A.2 for a definition.

For example, take the rule:

```
<doNothing, tails, strong, pos><*, tails, *, *> Pr: 1.0
```

The above rule is a more specific version of:

```
<doNothing, tails, *, pos><*, tails, *, *> Pr: 1.0
```

The second rule *subsumes* the first. If the extra condition has no significant effect on the probability of the rule then it is *covered* by the more general rule (and therefore unnecessary). In this example the additional condition *strong* has no significant effect.

More general operators are preferred because: they are more likely to apply to rules outside the original data set; a reduced number of rules can cover the same information; and the empirical probabilities of the leaned rules are more accurate (containing more samples).

The filter algorithm in this research adds an additional step to remove rule element sets that have no head. The ASDD algorithm (chapter 6) can generate rules without an outcome because it is searching for sets of rule elements that occur together, irrespective of whether an outcome is present. These partial rule element sets are of no further use after the generation stage.

```
filter (R)
   //sort R in non-increasing order of generality
   sortByGenerality(R);
   S = {};
   while (R != {})
      s = pop(R);
      push (S, s);
      for all (r ∈ R){
         if (head(r)={}) //remove if no outcome
            remove r from R;
         else if (subsumes(s, r) and gStatistic(s, r) < GLOBAL_G)
            remove r from R;
      }
   return S;
```

Algorithm 3-2: filter(R). R= complete set of candidate rule element sets.

- *R:* a set of rule element sets.
- *subsumes*(R_1, R_2): returns true if rule R_1 is a generalisation of R_2.
- g*Statistic*(R_1, R_2): returns the G statistic to determine whether the conditional probability of the outcome of R_1 given its conditions is significantly different from the probability of the outcome of R_2 given its conditions. The outcome is the same for both rules because they pass the *subsumes* test. This is, therefore, a test that the probability $o|R_1.x$ and the probability of $o|\ R_2.x$ are significantly different.

The constant *GLOBAL_G* is used as a threshold, which the G statistic must exceed before d_1 and d_2 are considered different. A value for *GLOBAL_G* of 3.84 tests for statistical significance at the 5% level, while a value of 2.706 tests for significance at 10% (used for smaller data sets).

See appendix section A.2 for explanation, pseudo-code, significance levels and explanation of the G-statistic. For a discussion of the issues relating to use of statistical significance in association rule mining, see Webb [96].

3.1.4 The Apriori Algorithm for Association Rule Mining

The Apriori algorithm addresses the problem of discovering association rules between items in a large database of sales transactions. An individual record in a database of this type generally consists of a *transaction date* and the *items bought* in the transaction (referred to as *basket* data). An example of an association rule is that 98% of customers purchasing tyres and car accessories also purchase a car service [1]. This can be written as a rule of the form:

$$\{tyres, car\ accessories\} \Rightarrow \{car\ service\}$$

Association rules find interesting relations between variables in the database. Two key forms of interesting relation are *support* and *confidence*.

Support is defined as the proportion of records in the database that contain the set of items in the rule. If 147 of the 7500 transactions in the database contain all three items (tyres, accessories and car service), then the rule has a support value of 1.96% (147/7500).

Confidence is defined as the probability that the conclusion of the rule follows the conditions. If there are 150 occurrences of transactions containing both tyres and accessories, then the probability of a car service being purchased at the same time is 147/150 = 98%.

The key feature of the Apriori algorithm for association rule mining is that it is able to generate and eliminate candidates of increasing complexity using less complex rules as a basis without the need to query the entire database for each new rule. It achieves this using the concept of *downward-closure*.

Downward-closure is the observation that for a frequent item-set, all of the subsets of the item-set must also be frequent. For an infrequent item-set, all of its supersets must also be infrequent. The Apriori algorithm exploits this property to avoid multiple passes over the transaction data when generating new rules. First, new item-sets of size *k* elements are generated by combining item-sets of size *k-1* for which all but one element are equal. Next the *k-1* size item-sets are searched to check that all subsets of each new size *k* item-set are

present. If any *k-1* size subset of a size *k* item-set is not present, then the item-set cannot have minimum support. Figure 3.1 shows this as a lattice of frequent item-sets.

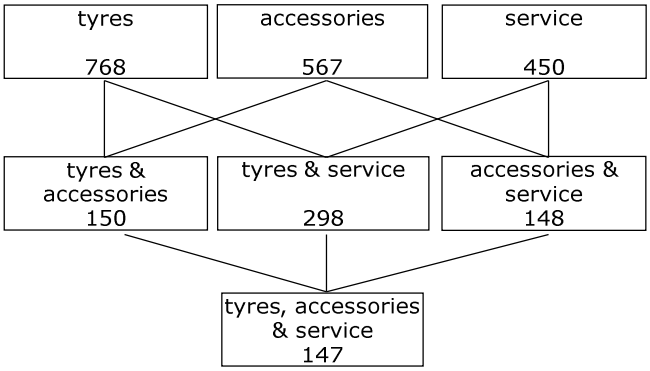


Figure 3.1: A lattice showing frequent item-sets with associated occurrences in a transaction database. The occurrence count of combined item-sets in the lower levels of the lattice cannot be higher than the minimum occurrences of a parent item-set.

Apriori and its descendants have been shown to scale up to large databases [41] and several adaptations have been developed for incrementally updating the learned rules. Typically these adaptations maintain a set of *fringe* rules for which additional data may provide evidence for inclusion to the set of significant rules [14]. For a survey of recent adaptations, see [4].

These features are highly desirable for model learning, with the need to process a potentially large database of perceptual data, and to incrementally improve the model as the agent receives new data.

The algorithm is the basis of Apriori Stochastic Dependency Detection (ASDD) and is explained in chapter 6.

3.1.5 Learning Noisy Deictic Rules (NDRs)

Pausla et al use an inductive logic programming (ILP) method to learn the noisy-deictic rules (NDRs) described in section 2.3.4 [67].

The algorithm uses three levels of search:

- *Learn Rules:* the outermost level, searches through the space of rule sets.
- *Induce Outcomes:* the middle level, constructs the outcome sets, given a context and an action.
- *Learn Parameters:* learns the probability of an outcome set.

Learn rules uses a greedy search in the space of *proper* rules. A rule set is defined as proper if every example in the data set has exactly one rule which matches it. Each item in the data set consists of a state and action followed by a successor state. Rules of this form can describe the data set because every effect that is possible given a context and an action is described in a single rule.

The search uses a heuristic scoring mechanism to rate the rule sets as they are generated by penalising rule sets with more complexity (conditions and effects) than are necessary, and scoring the set highly if it is able to reproduce the data (i.e. if the probability of a next state given a previous state in the data matches the probability generated by the rules).

The search is initialised by creating the most specific rule set: defined as a rule whose conditions are the state and action (*s, a*) for every pair in the data. This is a proper rule set as there is exactly one rule matching every state action pair from the data.

The output of the initial stage of the algorithm is, in effect, a tabular model of the form described in section 2.3. Each rule contains a value for every variable in the state in its conditions, and is followed by every possible successor state. A single rule for the coin flip example would be:

$$\begin{aligned} &flip(penny):\{\} \\ &\quad showing(penny, heads) \\ &\rightarrow \begin{cases} 0.5 : showing(penny, tails) \\ 0.5 : showing(penny, heads) \end{cases} \end{aligned}$$

The *showing(penny, heads)* condition is not relevant and will be removed by generalisation operators (below).

Learn rules proceeds by finding and applying an operator that will increase the score of the rule set (decrease the complexity while maintaining reproduction of the data) [54]. Four types of operators are used, based on operations used for rule search in inductive logic programming (ILP) [55].

The two generalisation operators are:

(i) Remove a condition from the rule.

(ii) Replace a constant with a variable.

The two specialisation operators are:

(i) Add a condition to the rule.

(ii) Replace a variable with a constant.

Once a new rule context has been created, a rule outcome is generated by finding the set of atoms which changed from the context to the outcomes (s_{t-1} to s_t) for each item from the data set matching the context. The set of outcome sets forms the basis of a proper outcome set for the rule.

This ILP based method can be used to create any possible rule set. However, as pointed out by Pasula et al [67], it suffers from the drawback that it is only guaranteed to create proper rule sets for the data in the training examples. Secondly, the "induce outcomes" method and heuristic rule scoring system are slow, each requiring a full pass over the data set for every new rule. The generated rules also have to include every possible outcome for a given rule, with the result that rules require an exponential number of outcome sets to the number of outcome atoms which have independent probability. In other words, the method is not able to learn parallel rules with independent outcomes (which can be learned using the ASDD algorithm presented in chapter 6).

3.2 Planning

Planning in artificial intelligence is the process by which an agent creates a plan that will take it from its current state to a goal state. In classical planning problems, the agent has an accurate deterministic model of an environment which it can use prior to execution to find a set of steps that will achieve a solution. The strategy can be formed using a search through the available actions and resulting states after execution of each action using a brute-force method, such as a depth first or breadth first search. The search space for a planning problem is, however, exponential to the number of variables in the state-space in the worst case, necessitating the use of search optimisation methods such as heuristic search (e.g. the A* algorithm [35]).

In stochastic environments, it is not possible to make a single plan and follow the steps to completion because actions can lead to non-deterministic successor states. If the random occurrences are caused by occasional interruptions in an otherwise deterministic environment, classical planning methods can be employed, with re-planning stages when an unexpected state is reached. Planning in inherently stochastic environments requires the formation of a *policy.* A policy is a universal plan which prescribes an action which should be taken in any state that the agent can reach. Rather than attempting to reach a single goal, a policy maximises the future rewards that an agent will receive in an environment.

If an environment model is known, policy formation can be achieved by using methods such as dynamic programming (referred to as decision theoretic planning [7] when used in the context of planning problems), or, more recently, Monte Carlo tree search [19]. If a model is not known, a policy can be formed by model-free methods such as temporal-difference learning [89], which achieve the same goal of attaching values to states, or state-action pairs. These techniques are collectively known as reinforcement learning methods because they aim to reinforce the selection of actions which lead to rewards [87]. The techniques can also be used when a model is known, and are often more efficient because they require less processing to update a state-value in a single iteration [73]).

3.2.1 Reward and Value

The problem facing a decision-making agent is to select the action which maximises its expected future rewards at each stage on the basis of a history of observations. In an environment displaying the Markov property, the next state is a function of the current state, so a history of size one contains equivalent information to any history of states for the decision maker.

A Markov Decision Process (MDP) is one which assigns value to taking each action in each state. In the MDP framework a *value function* estimates how beneficial it is for an agent to be in a given state (equivalent to the quality of taking the best available action in that state). The benefit of being in a state is defined in terms of future expected rewards.

- A *reward function,* $R : S \to R$, associates a reward, *r*, with being in a state, *s*.
- A *cost function,* $C : S \times A \to R$, associates a cost with performing action, *a*, in state, *s*.

In order to evaluate a course of action, it is necessary to define how many stages it will take to execute. If the course of action has an infinite number of stages it is known as an *infinite-horizon problem* or a *continuing task*. If the number of stages is discrete, it is known as a *finite-horizon problem* or an *episodic task.*

An example of an *episodic task* would be a chess game, in which there is a winner or a draw at the end of each episode. An example of a continuous task would be an investment agent tasked with maximising profit with no time limit. The agent's task is to continually pick an action which will maximise its expected future profit at each time step.

The value of a history (*h*) of observations of length T is defined for a *finite-horizon problem* as the sum of rewards gathered $R(s^t)$, at each stage, t, and costs incurred for each action at each stage $C(s^t, a^t)$ over the history. There is no action taken in the final, terminating, state,

but a reward is gathered, so the equation contains a final addition of reward in state s^T, with no associated cost.

$$V(h) = \sum_{t=0}^{T-1} \{R(s^t) - C(s^t, a^t)\} + R(s^T) \tag{3.1}$$

The value of an *infinite-horizon problem* may be unbounded. A common solution to this is to employ a *discount factor (γ),* where $\gamma < 1.0$, which ensures that rewards (and costs) at later stages are less than those at earlier stages. The expected value function for a discounted reward problem is defined as follows:

$$V(h) = \sum_{t=0}^{\infty} \gamma^t (R(s^t) - C(s^t, a^t)) \tag{3.2}$$

3.2.2 Solution Methods for Markov Decision Processes

The standard solution method for finding an optimal policy for a Markov Decision Process is *value iteration* (see below). Modern methods use value iteration as a basis, with approximate or efficient adaptations. For a discussion of the state of the art in current solution methods, see Powell [73].

Value Iteration

An optimal policy can be generated by repeatedly calculating the value of a state, based on the best action that can be taken in that state. This method is called *value iteration* [5][6] and works by feeding back rewards when they are received in particular states to refine the values of the state leading to the rewards. An optimal policy is one that picks the action with the maximum value from the current state calculated by summing the probabilities of going from state, *s*, to the set of possible states that the action can reach, multiplied by the rewards for each of these state (*s'*).

The value iteration algorithm works by making a series of approximations to the true value of the optimal policy by repeated application of equation (3.3). Value iteration can only be used if a full model (as opposed to sampling model) of the environment exists because it requires the probabilities of all successor states to be known. The update equation for value iteration is given by the Bellman equation [87]:

$$\begin{aligned} &V_{k+1}(s) = \max_a \sum_{s' \in S} P^a_{ss'} [R^a_{ss'} + \gamma V_k(s')] \\ &P^a_{ss'} = \text{probability of moving from state } s \text{ to } s' \text{ given execution of action } a \\ &R^a_{ss'} = \text{reward received when action } a \text{ is taken in state } s \text{ and leads to state } s' \end{aligned} \tag{3.3}$$

An alternative form can be used if rewards are dependent on taking an action in a state, rather than the action leading to a next state.

$$V_{k+1}(s) = \max_a (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_k(s'))$$
$$P_{ss'}^a = \text{probability of moving from state } s \text{ to } s' \text{ given action } a \quad (3.4)$$
$$R_s^a = \text{reward received when action } a \text{ is taken in state } s$$

The value of state *s* on pass *k +1* of value iteration is calculated by taking the maximum valued action. The value of the action is equal to the sum for all the action's following states, s' (where $s' \in S$), of the probability of the action leading from state *s* to *s'* (shown as $P_{ss'}^a$), multiplied by the discounted value of state *s'* on pass *k* (shown as $V_k(s')$), plus the reward for taking action *a* in state *s* (R_s^a). The discount factor, γ, must be less that 1.0 and is generally a number close to 1.0 (e.g. 0.95).

In order to generate a value map, the agent starts with a state generated at random and adds this to the value map. A single entry in the value map is stored as: {*State*, *Value}*. There is one entry for each state that the system can reach. If the agent has a complete value map, a policy can be generated by simply choosing the action with the highest value for that state.

Once the value map has been generated, a policy is equivalent to finding the maximum valued action in a given state.

3.2.3 Reinforcement Learning

If a model of the environment is not known in advance, in terms of either the state transition or the reward function, then reinforcement learning techniques can be used to acquire a state-action-value map from direct interaction with the environment. The standard reinforcement learning techniques are based around Q-learning (see below). For an introduction to reinforcement learning see Sutton and Barto [87].

Q-Learning

Q-Learning was introduced by Watkins [94][95]. The value of a state-action pair is calculated, rather than the value of the state itself. There is an efficiency benefit, in policy calculation and policy use for an agent, in that it is not necessary to calculate all the following states for every possible action in a current state in order to evaluate which action should be taken next. The agent need only search the available actions in the table for the given state and pick the maximum valued state-action pair.

The update function for Q-learning is as follows:

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha[R^a_{ss'} + \gamma \max_{a' \in A} Q_k(s',a') - Q_k(s,a)] \quad (3.5)$$

The equation contains a *discount factor (γ)* (defined previously in the Bellman equation), and a step size, *α*. This is an example of an update rule, where gradually improving estimates are made on a value function. The *α* parameter is a step size indicating how quickly the new estimate should change the old one. In Q-learning, the *α* parameter is known as the learning rate. If a step size of 1 is used, then the update is equivalent to the deterministic environment form of the Bellman equation (3.3). The equation step size must be in the range: $0 \le \alpha \le 10$.

The update rule is only necessary if a model of the environment is not known. In practice, however, computing all possible successor states with associated probabilities can be prohibitive in terms of processing requirements, and it is often only feasible to compute sample updates.

If the agent continually follows an optimal policy (picks the best action at each stage) with some error introduced in order to allow it to explore, the Q-learning algorithm will converge on an optimal policy with a probability close to 1.0 [87].

3.2.4 Learning Rate

The learning rate, *α*, for an update rule such as Q-learning does not have to be fixed. Often it is advantageous to use a high learning rate initially and then reduce the learning rate in the later stages, when the value estimates become stable. The initial Q-values will be a predetermined number (either zero or an arbitrary estimate of average value). Any update is likely to be an improvement on this initialised value, with the consequence that it is advantageous to use a learning rate close to 1.0 for the first update. Later estimates need to converge gradually, and require a low learning rate.

McClain's formula [58] was used for the learning rate in experiments in chapters 8 and 10. This is a deterministic formula which begins with a learning rate of 1.0, allowing maximum information to be extracted from the first iteration, and gradually decreases to a set minimum. The minimum has been set at 0.1 to keep the variance between iterations at a reasonably high level, while allowing convergence.

McClain's formula is given by:

$$\alpha_n = \frac{\alpha_{n-1}}{1 + \alpha_{n-1} - \bar{\alpha}} \quad (3.6)$$

$\bar{\alpha}$ is a parameter specifying the minimum step size (in this case 0.1) which the function will tend towards .With $\bar{\alpha}$ = 0.1, the step size is roughly 0.2 after two iterations and will be close to $\bar{\alpha}$ after ten iterations.

3.3 Summary

This chapter introduced model learning and planning techniques. The simplest model learning technique is a tabular method, which keep a record of every state encountered, action taken in the state and the frequency of each successor state reached. Given a state and an action the frequency of successor states can be read from the table to give a model of possible successor states.

Techniques of learning factored state models focussed on planning operator learning techniques, such as MSDD and an ILP method for learning noisy deictic rules (NDRs). The Apriori algorithm was also introduced, which will form the basis of the ASDD operator learning algorithm defined in chapter 6.

The concept of AI planning was then discussed, in the classical sense of forming a strategy for taking an agent from a current state to a goal state in a deterministic environment, and then in the decision theoretic planning sense of forming a universal plan, or policy, which provides an agent with an action to take in any reachable state in an environment.

4. Environment Modelling Agent Framework

This chapter presents the environment agent modelling and planning framework used in this research. The framework is built on the embodied agent environment model (section 2.1.4).

The agent's task is to build a perceptual model of its environment and then plan using this model. Initially the agent has knowledge of the actions it can perform, but not the effects, and has a *perceive* function that maps the sensory input it receives to a percept (performing basic pre-processing). The agent's task is to discover which elements of the percept are (stochastically) affected by its actions, the conditions under which these effects will occur, and an associated probability. The agent can use this model to develop a plan (or policy) to achieve its goals using reinforcement learning or dynamic programming techniques.

This section first introduces Dyna-Q [88], which acts as a starting point for the modelling framework, then defines the batch processed modelling, policy learning, and action framework used in this research.

The framework was previously presented by the author in outline form in *"SMART (Stochastic Model Acquisition with ReinforcemenT) Learning Agents: A Preliminary Report"* [17].

4.1 Integrated Planning, Acting and Learning

Figure 4.1 shows how the Dyna-Q framework defines the relationship between planning, acting and learning for an on-line learning process (adapted from [87]). This process uses experience from the environment to create a world model, and integrates real experience from the environment with simulated experience produced by the model as training data for a reinforcement learning algorithm.

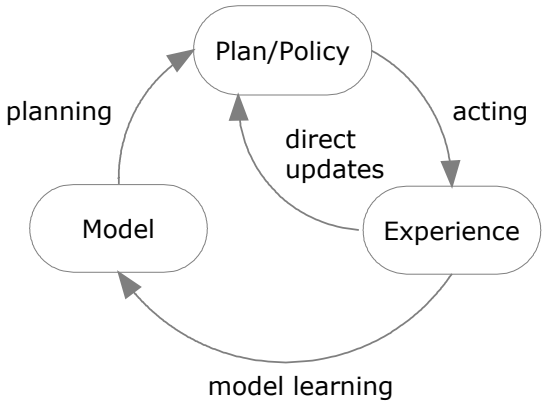


Figure 4.1: Integrated planning, acting and learning.

A planning agent can use experience to improve the accuracy of its world model and, simultaneously, to directly improve a value function via reinforcement learning. Simulated experience from the world model can be used in conjunction with new experience as it arrives to provide input to the reinforcement learning algorithm.

Algorithm 4-1 gives the Dyna-Q algorithm. The algorithm stores a tabular model, matching previously seen states and actions to successor states. If the environment is deterministic, each state-action pair will match to exactly one successor state. The algorithm can include a stochastic tabular model by recording successor states and frequencies and, when the model is queried, retrieving a sample successor state (see section 2.3).

The model can produce a biased Q-value map, because early experience and choices of action are repeated more often than later experience by continual execution of the model-based steps with each new experience gathered. The value of n determines the number of times the model is used to update Q for each additional input of real world experience. A high value for n can speed up convergence in environments with a limited state-space, but will bias Q-values towards the area of the environment explored initially in large state-spaces. A further issue is that the random model sample step will often be inefficient, updating areas of the Q-value map that contain no useful information because no reward state has yet been encountered in a trajectory from the initial or successor states. One solution to this is to use the *prioritised sweeping* technique, which updates only the parts of the model which can lead to a state that has previously changed value [60], although this has been shown to perform poorly in environments containing easily reachable sub-optimal solutions [39].

The *e-greedy(s,Q)* step provides exploration, by ensuring that the agent does not always take the current estimate of best action. A *greedy* action is one which takes the action with the highest Q-value for the current state, while an *e-greedy* action will, with a defined probability, e, take a random action. Initially the Q-value estimates will be inaccurate and it is important to ensure exploration occurs in the early stages (high probability of a random action), but that exploitation occurs when the estimates become stable (low probability of a random action). The e value can be fixed at a low probability (e.g. 0.1) to ensure this balance, or can be gradually reduced as the Q values stabilise. For a discussion of fixed and variable values for e see [87] and [73].

```
Dyna-Q
   while (true)
      s←current state
      a←e-greedy(s,Q)
      execute action a, observe following state s' and reward r
      Q(s,a)←Q(s,a)+α[r+γmax_a'Q(s',a')-Q(s,a)]
      model(s,a)←s',r   //update model
      repeat n times:
         s←random previously observed state
         a←random action previously taken in s
         s',r←model(s,a)
         Q(s,a)←Q(s,a)+α[r+γmax_a'Q(s',a')-Q(s,a)]
```

Algorithm 4-1: The Dyna-Q algorithm for deterministic environments (adapted from [87]). model(s,a) denotes the contents of the model. The steps before the model(s,a) step implement standard tabular Q-learning. The remaining steps implement model based learning.

4.2 Batch Processed Environment Modelling and Planning

Figure 4.2 shows how the framework used in this research adapts Dyna-Q to perform a batch process for environment model learning. The modelling and planning processes are separated, while taking action in the real world involves extraction of the best action for the given percept from the policy (universal plan). The separation of modelling and learning steps is necessary because the algorithm used to build the planning-operator-based world model is a batch process (see section 6). Extensions to the algorithm to include in-line operator learning and refinement are discussed in section 11.3. In-line operator learning would allow a standard Dyna-Q architecture to be employed. An advantage of separation of environment modelling and reinforcement learning steps is that the model can be re-used if the reward function is changed (see section 4.3).

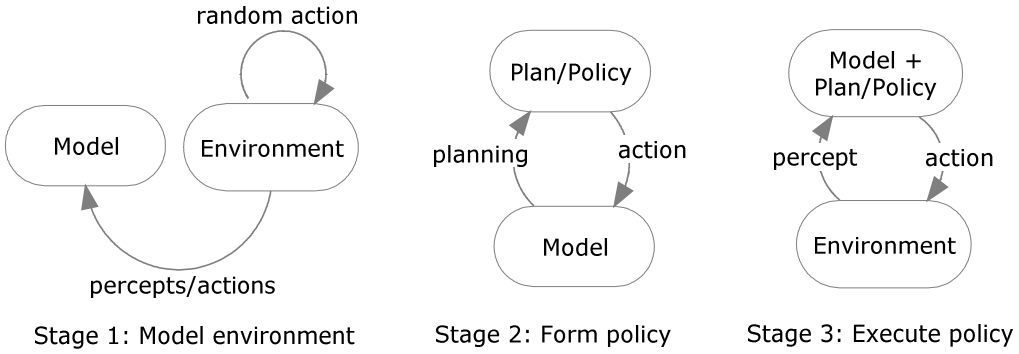


Figure 4.2: BatchModelQ. The process separates model learning and policy formation (planning) stages. The policy can be used to select actions in response to percepts received from the environment.

4.2.1 Stage 1: Model Environment

Initially, the agent takes random actions in the environment for a fixed period and receives percepts in response. The environment is assumed to be a discrete time environment and a percept is received in response to each action. The agent actions can include a "do nothing" action for a time step, which is treated in the same way as any other action and a percept is received in response.

A list of perceptual data items (PDIs) is stored, comprising the initial percept, action and successor percept (stored as the next initial percept to avoid repetition). A second list stores the rewards received at each time step. The PDI and reward list are next passed to a batch modelling process, such as the tabular model system defined in section 3.1, or the planning operator learning algorithm defined later in chapter 6.

The output of this stage is an environment model which the agent can use to simulate percepts it will receive in response to an initial percept and action.

```
batchModelQ-ModelEnvironment()
   initialise: model, M,
               perceptual data items, PDIs
               reward record, Rr
   p←initial percept
   a←generate random action
   add(PDIs,p,a);
   repeat n times {
      execute action, a, observe resultant percept & reward: p', r
      add(Rr, r);
      add(PDIs,p,a);
      a←generate random action;
   }
   learnModel(M, PDIs, Rr);
   return M;
```

Algorithm 4-2: batchModelQ-ModelEnvironment. The algorithm repeatedly takes random actions in an environment to build up a database of perceptual data items (PDIs). PDIs are used to learn a model via a batch learning algorithm.

Note that the model is initially empty and the model is learned in a single batch learning process. A simple, model agnostic, method of providing background knowledge to the system would be to provide an initial PDI and reward set, which can then be built upon using real-world experience. A similar method could also be used to provide a model agnostic method of on-line learning by generating a sample set of PDIs and rewards to feed into the next model from previous models. On-line methods of updating the planning-operator-based model used in this research are discussed in section 11.3.3.

4.2.2 Stage 2: Form Policy

The second stage of the *BatchModelQ* process enables the agent to generate a policy using the acquired model to provide simulated experience.

A standard tabular reinforcement learning algorithm can use the model to simulate experience and learn a table of state-action values. In this case, values can be associated with each state-action pair in the model.

Alternatively, the RVRL algorithm can be employed to associate approximate state-values with a rule-based model (chapter 9). Values are stored in the model using the function *store(M,p,a,v)* and retrieved using *retrieve(M,p,a),* where *M* is the model, *p* is the percep, *a* is

the action, and *v* is the updated state-action value (or alternatively state value using a similar function with the *a* parameter omitted).

Form Sample Policy

Algorithm 4-3 shows how estimates of state-action value can be iteratively improved by using simulated successor percepts and rewards in response to an input percept and action. The use of sample updates and a standard reinforcement learning algorithm requires that an *e-greedy* action selection mechanism be used in order for the agent to explore. Each update uses a Q-Learning update function (section 3.2.3), which includes, α, an update speed parameter, because the algorithm is continuously improving estimates of the Q-values.

```
batchModelQ-formSamplePolicy(M, p, a)
   repeat n times {
      p',r←sampleModel(p,a);
      Qval = retrieve(M,p,a);
      Qval = Qval+α[r+γQ_k(p',a')−Qval];
      store(M,p,a,Qval);
      p←p';
      if (random < e-value)
         a←random action;
      else //retrieve best Qval action for p
         a←greedyAction(M,p);
   }
```

Algorithm 4-3: batchModelQ-formSamplePolicy. M = the environment model, p = an initial percept, a = an initial action. The algorithm uses reinforcement learning to update values in the model from sample successor percepts and rewards.

Form Distribution Policy

Algorithm 4-4 shows how a policy can be formed by using a distribution model to generate a set of simulated successor percepts and rewards with associated probability in response to input percepts and actions. The use of a distribution model update function is similar to the Bellman update introduced in section 3.2 but updates state-action pairs, rather than state values. There is no need to use an update speed, α, because full updates are used (rather than e-greedy action based sample updates) resulting in the utilities being distributed according to the probabilities provided by the model. A value of between 0.9 and 0.95 is typically used for γ (discount for future rewards) in systems of this type. The algorithm is similar to value iteration [87], but evaluates random percept-action pairs (or equivalently state-action pairs for fully-observable environments), rather than evaluating states in order. The algorithm also

attaches values to percept-action pairs rather than percepts (analogous to using the post-decision state variable in approximate dynamic programming [73]).

```
batchModelQ-formDistributionPolicy(M, p, a)
  repeat n times {
    {p'1,r1,pr1,...,p'n,rn,prn}←distributionModel(p,a);
    newValue = Σ_{i=p'1 to p'n} pri[ri + γ max_a' retrieve(M,p'i,a')];
    store(M,p,a,newValue);
    p←random selection from {p'1,...,p'n};
    a←random action;
   }
```

Algorithm 4-4: batchModelQ-formDistributionPolicy. M = the environment model, p = an initial percept, a = an initial action. The algorithm uses dynamic programming to update state-action values in the model from the set of successor percepts and rewards.

Note that the performance of the algorithm could be improved by using a prioritised-sweeping-based method to choose state action pairs whose values have previously been updated to avoid wasteful updates of unexplored areas of the value map (as discussed in the Dyna-Q algorithm above).

Algorithm 4-5 shows how Bellman updates can be used to form a policy using a distribution model. The algorithm has an amended update function, associating values with states (or equivalently percepts), rather than a state and action. The form of the algorithm is equivalent to a standard Bellman update with the maximum action being found through the *greedyAction* function (see *stage 3* below). The algorithm is equivalent to using value iteration with a fixed number of steps.

```
batchModelBellman-formDistributionPolicy(M, p)
   repeat n times {
      a←greedyAction(M,p);
      {p'1,r1,pr1,...,p'n,rn,prn}←distributionModel(p,a);
      newValue = Σ_{i=p'1 to p'n} pri[ri + γ retrieve(M,p'i)];
      store(M,p,newValue);
      p←random selection from {p'1,...,p'n};
   }
```

Algorithm 4-5: batchModelBellman-formDistributionPolicy. M = the environment model, p = an initial percept. The algorithm uses bellman updates to update state values in the model.

4.2.3 Stage 3: Execute Policy

The third stage of the process enables the agent to use its policy to select actions in response to percepts received from the environment. Stage 2 associated a value with each state, or state-action pair accessible via the model. Execution of a policy means selecting the action with the highest value from those available in response to the current percept.

Algorithm 4-6 shows how the highest value action can be extracted from the model.

```
greedyAction(M,p)
   maxAction = null;
   maxActionValue = 0;
   for all a ∈ A {
         actionValue = retrieve(M, p, a);
         if (actionValue >= maxActionValue) {
            maxActionValue = actionValue;
            maxAction = a;
         }
   }
   if (maxAction == null) //no model stored for p
      action = random action;
   return maxAction;
```

Algorithm 4-6: greedyAction. M = the environment model with associated values, p = an initial percept. The algorithm returns the highest valued action available for the percept.

4.3 Discussion of Model-Based Reinforcement Learning

The main advantage of a model-based learning technique is that it often requires reduced real-world experience to form an effective policy in comparison to standard reinforcement learning. With standard reinforcement learning, value updates are only passed back from a single state, or state action pair, to the preceding one. The process of learning a policy requires repeating action sequences several times before the knowledge is acquired by the agent.

Models can also be used in conjunction with approximate dynamic programming methods to form compact policies. Dynamic programming in a stochastic environment requires full backups and cannot be used in either a model free context or a sampling model context. Distribution model techniques can only be used if such a model exists, or can be acquired through experience.

A model of the dynamics of an environment can be re-used when the reward changes. A predator with an effective model of the dynamics of its environment can use the model if its

reward function changes (e.g. it becomes a prey). An example of this would be the ghosts in the arcade classic Pac Man, that begin the game as predators, but become prey when the Pac Man eats a pill. A policy learned directly from the environment becomes useless in this situation, while one learned from the model can be re-learned from the same model by altering the reward function.

The separation of model learning, planning, and acting phases used in this research provides a useful framework to simplify the modelling process for the agent, but this is not a necessary restriction for the framework in general. An in-line process could be used to improve the model as new experience is acquired, and reinforcement learning can be performed to improve value estimates using the improved model.

If the acquired model is a true representation of all experience learned then it should not be necessary to integrate real experience (as used in the Dyna-Q framework). The main advantage of the real experience steps is that recent experience can have a greater influence on the policy learned if the environment is not static [87]. This could also be achieved by biasing the model learning process towards more recent experience using, for example, a Bayesian update function for probabilities [68].

The main disadvantage of environment modelling methods over direct reinforcement learning is that errors in the design of the model, or a bias in the sample used to make the model, can cause incorrect simulated experience. The methods also increase the complexity of the learning algorithm through the additional model learning process.

A disadvantage of the batch modelling method is that random actions are taken in the modelling phase. This method of environment modelling can be problematic if parts of the environment are only accessible through guided action. For example, the chances of winning an adversarial game through random action can be vanishingly small (e.g. winning chess against a grand master). This disadvantage is mitigated to some extent by modelling the environment through planning operators (e.g. it may be possible to learn the rules of chess through random action, even if a winning state is never achieved).

4.4 Perceptual Environment Modelling

The separation between agent and environment through an agent body (section 2.1.4) requires that the agent body contains sensors, which are updated as part of the environment update cycle and represent direct measurements of the environment state (including the agent body). The agent's *perceive* function converts sensor data into a percept. The perceive function can

be a one-to-one mapping from sensor feature to perceptual feature, or can include some pre-processing to convert the percept into a more useful form for the environment modeller.

Chapter 5 demonstrates the use of the perceive function to convert the sensor data into axioms, which form the basis of parallel stochastic planning operators. Other uses of pre-processing through perception could include extraction of object data from raw image files.

The agent's model of the environment is at the percept level, rather than the sensor level. This means that it is modelling the expected percept it will receive in the following time step in response to an action taken. The agent does not need a function to convert the percept back to sensor data, because its environment model is at the percept level and it can make decisions based on predictions of future percepts.

4.5 Summary

This chapter presented a batch process framework for agent environment modelling and policy formation. The agent selects random actions in an environment and receives percepts and rewards in response. The record of percept, action, reward is used to build a model of the environment. A policy learning stage uses the model to simulate environment experience and reinforcement learning or dynamic programming is employed to update value estimates stored in the model. Finally, values stored in the model can be used to select optimal actions in response to an input percept or state.

5. Parallel Stochastic Planning Operators: P-SPOs

This chapter defines the syntax of Parallel Stochastic Planning Operators (P-SPOs) and the associated algorithms for generating successor percepts given a current percept and action. P-SPOs are used as an environment model in the agent framework presented in section 4.1. The operators are designed with the aim of enabling automated acquisition from experience, but they may alternatively be designed by hand.

The operators are a *parallel* extension of Noisy Deictic Rules (NDRs) [67], allowing more than one operator to be applied in a given time step to generate a successor state. The representation is powerful in terms of modelling an environment and the result of an agent's action within it, because operators can express *independent* as well as *conditional* outcomes. An outcome is independent if its probability is not affected by the value of any other outcome and conditional otherwise. The syntax has a structure that facilitates acquisition from data (chapter 6) and the representation can be used to group parts of a state (or percept) space for a state-aggregation based value map (chapter 9).

P-SPOs model a mapping from precursor to successor percepts in response to an action. Each percept contains a number of perceptual features, which may be a direct mapping from sensor data, or a more complex pre-processing provided by the agent *perceive* function. The available actions and perceptual features of the environment can be provided as background knowledge, or acquired through the P-SPO learning process.

During planning, each perceptual feature can take only one value in each simulated future percept. Probabilistic outcomes are modelled as a set of fully realised future possible percepts, rather than percepts with probability distributions over perceptual features. E.g. a Boolean perceptual feature will be either *true* or *false* in each possible future percept.

As discussed in section 2.2, sample perceptual models take the form:

$$samplePerceptualModel : p \times a \rightarrow p'$$

A distribution model returns a set of successor percepts with associated probabilities:

$$distributionPerceptualModel : p \times a \rightarrow \{\{p_1, pr_1\}, \{p_2, pr_2\}, \ldots, \{p_n, pr_n\}\}$$

$$where \sum_{i=1}^{n} pr_i = 1.0$$

This chapter first defines the syntax of P-SPOs, followed by algorithms for successor percept generation with worked examples, and finally defines *environment* operators, which enable compact operator set representation in stochastic environments.

5.1 Introduction

Parallel Stochastic Planning Operators model changes to an environment in response to actions through the use of *outcome sets*. The outcomes define the changes to perceptual features (as opposed to adding or deleting elements).

Each Parallel Stochastic Planning Operators has:

- *An action*: one of the available actions selectable by the agent; the special *environment* operator (used to model the action of the environment); or empty, {}, indicating that the outcomes are independent of agent or environment action.
- *A context*: a (possibly empty) set of conditions which determine when the operator can be applied.
- *An outcome set, with associated probability*: determines the expected value of perceptual features in the successor percept if the action is applied in the context. Each outcome has an associated probability, with a sum of 1.0 for the complete outcome set.

P-SPOs are parallel in that more than one operator can be applied in the same time step. Operators can be defined with conflicting outcome sets, but may not be applied in parallel if there is a conflict. A further restriction on parallel operator application is that at each time step only one action can be selected by the agent.

This parallel extension provides the ability to model multiple independent outcomes with a minimal set of operators. An outcome conflict occurs when the outcome sets of two applicable operators refer to the same perceptual feature. If a conflict occurs then operator precedence is applied. Operator precedence (section 5.4.4) provides a conflict resolution technique that will, in general, favour the most specific operator applicable to the current world state, but is generated using a heuristic over conflicting data sets (section 6.6). All actions can be attempted in all environment states. An action which has no effect in a particular state has an empty outcome set.

5.2 Syntax

Parallel stochastic planning operators and the percepts to which they are applied are defined using a restricted form of standard first-order logic that does not include negation, disjunction or existential quantification. Functions are included but are restricted to immutable background knowledge. Negation is not included, with a preference for Boolean axioms, which have the same expressive power but allow generalization to multi-valued variables.

The syntax is a parallel adaptation of that used in Probabilistic Relational Planning Rules [66][100] and Noisy Deictic Rules (NDRs) [67].

In the following sections an example of a block-painting robot is used. The robot is able to observe several features of its environment that are relevant to its task. This is known as the "slippery gripper" problem as adapted in [65]. Figure 5.1 is a representation of the environment, which consists of:

- *A block:* There is exactly one block at all times. The block can be painted or unpainted.
- *A gripper:* There is exactly one gripper, which can be dry or wet. The gripper can be holding the block.
- *A reward:* There is a positive reward each time a painted block is delivered, a negative reward each time an unpainted block is delivered, and no reward otherwise.

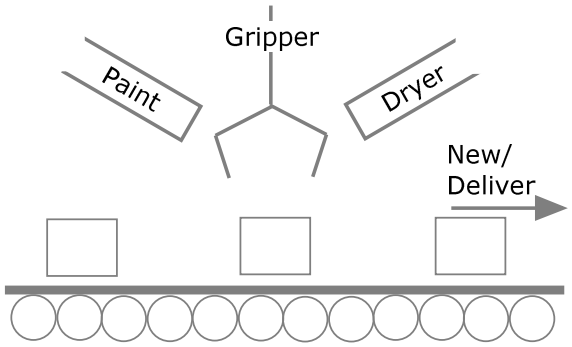


Figure 5.1: The "slippery gripper" environment. The robot's task is to paint blocks which arrive on a conveyer belt, and deliver the blocks once painted.

5.2.1 Percept and State Representation

The following section defines the percept representation for P-SPOs. Percept representation is equivalent to state representation in a fully observable environment.

The percept description below (5.1) represents the painting robot's perception of its environment when the block, *b*, is not painted (*painted(b, false)*), the gripper, *g*, is wet *(dry(g, false)*), the robot is holding the block (*holding(b, true)*) and it received no reward in the previous time step (*reward(none)*). The robot has four perceptual features. The first three (*painted*, *dry* and *holding*) are Boolean, while the fourth, *reward*, can take three values, *pos, neg* and *none*.

$$painted(b, false), dry(g, false), holding(b, true), reward(none) \quad (5.1)$$

The agent's percept is represented by a conjunction of positive ground literals encoding the value of all perceptual elements. Each perceptual element can take on one of a set number of values defined by background knowledge (provided or acquired empirically). Constants map to all observable elements in the percept, while literals, encode positively observed properties of environment features. A percept is a concrete instantiation, giving a finite set of observed features, a finite set of literals, and is described by a set of positive ground literals.

Negation is not included to allow a concise description of perceptual features without recourse to negation by failure. If a state has to be fully defined using negation, each percept would have to contain all elements that are not currently observed as well as those that are. This is a particular issue for non-Boolean variables. If, for example, the robot's current reward is *none*, a percept definition (without negation by failure) would have to state that the robot's current reward is not *pos* and not *neg*.

In the current example the percept could be defined with an implicit gripper and block because there is only one gripper and one block. The operators are presented with defined gripper and block because this representation allows the flexibility to add additional blocks.

5.2.2 Background Knowledge

Background knowledge for an environment defines:

- The possible values of the perceptual elements.
- A set of conflicts, which are used to restrict the simulated successor percepts to contain a valid set of perceptual features.

The function *conflicted(P)* takes a percept, *P,* and returns *true* if the percept is in a conflicted state.

Figure 5.2 gives an example of the background knowledge for the "slippery gripper" environment.

$$
\begin{array}{l}
reward(X) \leftarrow X \in \{pos, neg, none\} \\
boolean(X) \leftarrow X \in \{true, false\} \\
painted(X,Y) \leftarrow block(X), boolean(Y) \\
dry(X,Y) \leftarrow gripper(X), boolean(Y) \\
holding(X,Y) \leftarrow block(X), boolean(Y) \\
conflicted(P) \leftarrow painted(X, true) \in P,\ painted(X, false) \in P \\
conflicted(P) \leftarrow dry(X, true) \in P,\ dry(X, false) \in P \\
conflicted(P) \leftarrow holding(X, true) \in P,\ holding(X, false) \in P \\
conflicted(P) \leftarrow reward(X) \in P,\ reward(Z) \in P,\ X \neq Z
\end{array}
$$

Figure 5.2: Background knowledge for the "slippery gripper" environment.

Background knowledge is represented as set of logic rules defining properties of the observable features of the environment. A percept is said to be valid if there are no conflicts. Conflicts define the possible values of each perceptual feature.

5.2.3 Parallel Stochastic Planning Operator Representation

P-SPOs model both the effects of agent actions on an agent's expected next percept and external changes caused by the environment. If the agent has direct access to the environment state (a fully observable environment) then the P-SPOs model is equivalent to a state model.

A P-SPO set is a set of operators.

- Each $pspo \in P\text{-}SPOs$ is a five-tuple, $<P_A, P_C, P_O, P_{Pr}, P_D>$.
- P_A: the operator action is a positive literal, with a predicate representing the action, and terms representing constants in the percept. The action may be empty (shown as {}).
- P_C: the operator context is a conjunction of positive literals, or empty.
- P_O: a set of outcomes $\{P_{O1}, \ldots, P_{On}\}$ where each outcome is a set of positive literals that define the possible values of percept elements in the successor percept.
- P_{Pr}: a set of probabilities $\{P_{Pr1}, .., P_{Prn}\}$ associated with P_O, giving the probability that each of the outcomes, P_O, will occur. Variables in the outcome set must be present in the action parameters or context of the operator in order for resolution to be possible (and for the operator to be valid).
- P_D: a set of P-SPOs that this rule defers to in situations for which they are in conflict. This set can be inferred empirically when the operators are learned from data (see section 6.6)

Variables are denoted with capital letters. Constants, functions and literals are denoted by characters or strings with an initial lower case letter.

The subset of the P-SPO set for the *paint* action in the slippery gripper domain is shown below:

$$paint(X,Y):holding(Y,true)\rightarrow\{1.0:painted(Y,true)\}$$
$$paint(X,Y):painted(Y,false),holding(Y,false)\rightarrow\begin{Bmatrix}0.1:painted(Y,true)\\0.9:painted(Y,false)\end{Bmatrix}$$
$$paint(X,Y):dry(X,true)\rightarrow\begin{Bmatrix}0.4:dry(X,true)\\0.6:dry(X,false)\end{Bmatrix}$$

Each operator has:

- An action: e.g. *paint(X, Y).*
- A context: e.g. *holding(Y, true).*
- A set of outcomes with associated probabilities: e.g. *{0.1: painted(Y, true) 0.9:painted(Y, false)}*

Figure 5.3 gives the full P-SPO set for the "slippery gripper" domain. The P-SPOs describe an environment in which the robot's actions are:

- ***Paint:*** paints blocks. It results in a painted block if the robot is holding the block in its gripper 100% of the time. If the robot is not holding the block, the block has a 10% chance of becoming painted. The gripper will become wet 40% of the time.
- ***Dryer:*** dries the gripper. It results in a wet gripper becoming dry 90% of the time.
- ***Pickup:*** picks up blocks. It results in the block being held if the gripper is dry. The block will be held if it was not held and the gripper was wet 60% of the time.
- ***New****:* used to deliver a block and receive a new one. It results in a positive reward if the block was painted and a negative one if it was not. A new block arrives which is not painted or held in the gripper.

$$paint(X,Y): holding(Y,true) \rightarrow \{1.0: painted(Y,true)\} \quad (1)$$

$$paint(X,Y): \begin{matrix} painted(Y,false), \\ holding(Y,false) \end{matrix} \rightarrow \begin{Bmatrix} 0.1: painted(Y,true) \\ 0.9: painted(Y,false) \end{Bmatrix} \quad (2)$$

$$paint(X,Y): dry(X,true) \rightarrow \begin{Bmatrix} 0.4: dry(X,true) \\ 0.6: dry(X,false) \end{Bmatrix} \quad (3)$$

$$dryer(X): dry(X,false) \rightarrow \begin{Bmatrix} 0.9: dry(X,true) \\ 0.1: dry(X,false) \end{Bmatrix} \quad (4)$$

$$pickup(X,Y): \begin{matrix} dry(X,true), \\ holding(Y,\_) \end{matrix} \rightarrow \{1.0: holding(Y,true)\} \quad (5)$$

$$pickup(X,Y): \begin{matrix} dry(X,false), \\ holding(Y,false) \end{matrix} \rightarrow \begin{Bmatrix} 0.6: holding(Y,true) \\ 0.4: holding(Y,false) \end{Bmatrix} \quad (6)$$

$$new(Y): painted(Y,false) \rightarrow \{1.0: reward(neg)\} \quad (7)$$

$$new(Y): painted(Y,true) \rightarrow \{1.0: reward(pos)\} \quad (8)$$

$$new(Y): painted(Y,\_) \rightarrow \{1.0: holding(Y,false)\} \quad (9)$$

$$new(Y): painted(Y,\_) \rightarrow \{1.0: painted(Y,false)\} \quad (10)$$

$$environment: \{\} \rightarrow \{1.0: reward(none)\} \quad (11)$$

Figure 5.3: The P-SPO set for the "slippery gripper" environment.

The last two P-SPOs referring to the *new* action contain the *painted(Y,_)* condition. This ensures that the *Y* variable is only matched to blocks. The "_" indicates a "don't care" condition to allow the operator to be matched, irrespective of whether the block is painted. The *painted(Y,_)* could be replaced by a *block(Y)* condition, but this would require additional background knowledge to define blocks. Using the method above ensures that all conditions can be learned from the data present in a percept. These rules could have, equivalently, included the *holding(Y,_)* condition. Rule (5) shows an example of this for the pickup action.

The P-SPOs also contain an example of an environment operator (an operator that defines the evolution of a percept element if no action is applicable). An environment operator for the "slippery gripper" domain is:

$$environment: \{\} \rightarrow \{1.0: reward(none)\}$$

A full explanation of environment operators is given in section 5.5. This environment operator provides a similar mechanism to the frame assumption in that models perceptual features that are not affected by actions. The environment operator above tells us that the reward will change to *none* (or remain as *none*) if no other action has affected the perceptual element. This is a more powerful mechanism than the frame assumption, which would keep the value of each element the same (e.g. *reward(pos)* would remain as *reward(pos)* until an action changed it). Environment operators can also be used to contain important state-value

aggregation information for the Rule Value Reinforcement Learning (RVRL) system (chapter 9).

5.2.4 Dependent Outcomes

Figure 5.3 gave an example of the operators required to describe an environment in which all outcomes exhibit conditional independence. The addition to the representation needed to include dependencies between outcomes is defined below.

Dependencies between outcomes are modelled by P-SPOs as operators with multiple perceptual features in each outcome. If a perceptual feature is present in one outcome it must be present in all outcomes for that operator.

An example of this would be to alter our environment such that the gripper always becomes wet if we use the paint action and the block becomes painted. This can be by achieved by altering the operator set as follows:

$$paint(X,Y): holding(Y,true) \rightarrow \{1.0: painted(Y,true)\} \quad (1)$$

$$paint(X,Y): holding(Y,true), dry(X,\_) \rightarrow \{1.0: dry(X, false)\} \quad (1a)$$

$$paint(X,Y): \begin{matrix} painted(Y, false), \\ holding(Y, false) \end{matrix} \rightarrow \begin{Bmatrix} .1: painted(Y,true) \\ .9: painted(Y, false) \end{Bmatrix} \quad (2)$$

$$paint(X,Y): \begin{matrix} painted(Y, false), \\ holding(Y, false), dry(X,true) \end{matrix} \rightarrow \begin{Bmatrix} .1: painted(Y,true), dry(X, false) \\ .36: painted(Y, false), dry(X,true) \\ .54: painted(Y, false), dry(X, false) \end{Bmatrix} \quad (2a)$$

$$paint(X,Y): \begin{matrix} painted(Y,true), \\ holding(Y, false), dry(X,true) \end{matrix} \rightarrow \begin{Bmatrix} .4: dry(X, false) \\ .6: dry(X,true) \end{Bmatrix} \quad (3a)$$

Figure 5.4: Update to operators in the "slippery gripper" domain with additional dependencies between outcomes (the gripper always becomes wet if the block is painted).

In the operator set above:

- P-SPO (1a) has been added to indicate that the gripper becomes wet, *dry(X, false)*, as a definite result of the *paint* action if the robot is holding the block. This is not dependent on whether the gripper was dry previously and there is, therefore, no need to add a *dry(X, true)* condition. The variable *X* in the outcome does, however, need to be matched with an element of the conditions (or action) which is achieved using the *dry(X, _)* condition ("_" indicates "don't care" and is matched irrespective of whether it is wet or dry).
- P-SPO (2a) has been added with the condition *dry(X, true)* and the outcome set includes *dry(X, false)* for outcome sets in which the block is painted. *dry(X, true)*

is present if the block is not painted with probability 0.4 as with the previous rules. The probability of the combined outcome is, therefore, 0.36 (after multiplication by the 0.9 chance of the block remaining unpainted). Similarly the probability for the combined outcome *painted(Y, false), dry(X, false)* is 0.54.

- P-SPO (3a) is required to cover the situation when the gripper is dry and the block is painted. This can occur after a block is pained, and the *dry* action is used. The original rule (3) does not cover this because it will also match the situation when the paint action is used and the block becomes painted, altering the probability that the gripper will become wet.

5.2.5 Single Action Restriction

P-SPOs are restricted such that only one action can be taken in a single time step. This has the consequence that variables in the action definition can only be resolved once for a single action. If, for example we were working in a world with two blocks, *b1* and *b2,* the variables in the pickup action can only be matched once. The operators are:

$$pickup(X,Y):dry(X,true),holding(Y,\_)\rightarrow\{1.0:holding(Y,true)\} \quad (5)$$

$$pickup(X,Y):dry(X,false),holding(Y,false)\rightarrow\left\{\begin{matrix}0.6:holding(Y,true)\\0.4:holding(Y,false)\end{matrix}\right\} \quad (6)$$

An example state for a world with two blocks is:

painted(b1,true), holding(b1, false), painted(b2, false), holding(b2, false), dry(g, false), reward(none)

The single action restriction allows the following action variable resolution to be valid:

$$pickup(g,b1):dry(g,false),holding(b1,false)\rightarrow\left\{\begin{matrix}0.6:holding(b1,true)\\0.4:holding(b1,false)\end{matrix}\right\}$$

A second resolution (below) is also valid, but they could not both be resolved in the same time step. These resolutions therefore represent two separate actions.

$$pickup(g,b2):dry(g,false),holding(b2,false)\rightarrow\left\{\begin{matrix}0.6:holding(b2,true)\\0.4:holding(b2,false)\end{matrix}\right\}$$

Note that the operator set defined allows more than one block to be held in a gripper with no change to the probabilities. This is a slightly unnatural situation, but will suffice to keep the explanation simple. A more natural definition would include an extra condition in the *holding* percept to define which gripper was holding the block, and would have additional conditions

on the pickup action operators to alter the probability of a successful pickup if the robot is already holding a block in the gripper.

5.2.6 Action Parameters

Actions are selected explicitly by the agent, which has the consequence that all variables are instantiated before attempting to match conditions in the environment. This can be demonstrated by examining the *new* operators. The *new* action can be performed on block *b1* or block *b2*, but not both simultaneously.

Selecting the *new* action for *b1* gives:

$$new(b1) : painted(b1, true) \rightarrow \{1.0 : reward(pos)\}$$
$$new(b1) : block(b1) \rightarrow \{1.0 : holding(b1, false)\}$$
$$new(b1) : block(b1) \rightarrow \{1.0 : painted(b1, false)\}$$

Selecting the *new* action for *b2* gives:

$$new(b2) : painted(b2, false) \rightarrow \{1.0 : reward(neg)\}$$
$$new(b2) : block(b2) \rightarrow \{1.0 : holding(b2, false)\}$$
$$new(b2) : block(b2) \rightarrow \{1.0 : painted(b2, false)\}$$

The parameter *Y* could not be set to *b1* and *b2* simultaneously. This is not overly restrictive in terms of syntax, because an additional *new* action could be defined that allows the delivery of two blocks in a single time step. Care must be taken when creating (or learning) operator definitions that the same variable is used to denote the same element between action operators.

5.3 Successor Percept Generation

In order to build a complete successor percept, the value of each perceptual feature must be determined using the P-SPOs. The frame assumption determines that each perceptual feature of the successor percept will be unchanged for the current percept unless it is contained in the outcome set of a matching P-SPO (including the environment operator). The starting point for our successor percept is, therefore, a copy of the current percept. In situations where more than one rule can be applied to the same successor percept element, a conflict resolution must be applied via precedence between operators (section 5.4.4). P-SPOs, including *environment* operators, have defined precedence between them and all operators have precedence over the frame rule.

If the frame assumption was not employed, and there was no recourse to environment operators, P-SPOs would have to explicitly define the values of successor variables unaffected by an action for each action using rules such as:

$$pickup(X,Y):dry(X,true)\rightarrow\{1.0:dry(X,true)\}.$$

5.3.1 Generate a Sample Successor Percept

`generateSamplePercept` takes as input, a percept *p*, an action *a*, and the set of P-SPOs and returns a sample successor percept. This function can be used to generate an output for the *samplePerceptualModel* function of the framework presented in chapter 4.

```
generateSamplePercept(P, A, PSPOs)
   ResolvedMatchingPSPOs = matchingAndResolved(PSPOs,P,A);
   filterByPrecedence(ResolvedMatchingPSPOs);
   do {
      OutputPercept = P;
      for (Pm ∈ ResolvedMatchingPSPOs)
         applyBySample(OutputPercept, Pm);
   } while (invalid(OutputPercept));
   return OutputPercept;
```

Algorithm 5-1: generateSamplePercept. P= initial percept, A = action, PSPOs = P-SPO set. A sample percept is returned.

The algorithm first finds all resolved operators matching the percept, *P*, and action, *A*. E.g. if the percept contained *dry(g,true)* and the action is *pickup(g,b)* then the relevant matching operator is:

$$pickup(X,Y):dry(X,true),holding(Y,\_)\rightarrow\{1.0:holding(Y,true)\}$$

The resolved operator can be found by resolving for the variables *X* and *Y* giving:

$$pickup(g,b):dry(g,true),holding(b,false)\rightarrow\{1.0:holding(b,true)\}$$

Again, the "_" can be matched to either *holding(b,false)* or *holding(b,true)*.

Next, conflicting P-SPOs are removed using `filterByPrecedence` (section 5.3.3). This function checks for conflicts between outcome sets of every resolved matching P-SPO. If a conflict is found, the operator with precedence is retained, while the deferring operator is removed. The output of this step is a set of resolved matching operators with no conflicts in the outcome sets.

The successor percept is generated by applying each operator in turn to a copy of the input percept. The copy of the input percept retains all input perceptual features unless they are changed by an operator, thus implementing the frame assumption. A sample successor is generated by taking a random sample output from the output set, P_O, of each P-SPO.

A final check is made to ensure that the generated output percept is not in an invalid state (section 5.4.5). If this is the case, new percepts are generated until a valid successor is created.

5.3.2 Generate all Successor Percepts and Probabilities

The *generatePerceptsAndProbs* function generates a full set of possible successor percepts with associated probabilities. A set of successor states is built by applying each outcome for each operator in turn to the initial percept. If an operator has more than one outcome, the outcome set will generate multiple intermediate percepts, which form the input for the next operator.

Again, a final check is made to ensure that the generated output percepts are not in an invalid state using background knowledge (section 5.4.5). If percepts are found to be in an invalid state, these are removed from the output set and the probabilities of the remaining successor percepts are normalised. The function can be used to generate output for the *distributionPerceptualModel* function of the agent framework presented in chapter 4.

```
generatePerceptsAndProbs(P, A, PSPOs)
   Current = {};
   ResolvedMatchingPSPOs = matchingAndResolved(PSPOs, P,A);
   filterByPrecedence(PSPOs);
   push(Current, {P, 1.0}); //push percept and probability
   for (PSPO ∈ PSPOs){
      Next = {}; //empty stack for the intermediate precepts
      while ({PerceptItt, Prob} = pop(Current){
         //add a list of percept and probability pairs
         //after application of each output from PSPO.P_O
         push(Next, applyAllOutcomes(PerceptItt, PSPO, Prob));
      }
      Current = Next;
   }
   //finally, check for invalid states
   for ({Percept, Prob} ∈ Current){
      if (invalid(Percept))
         remove(Percept, Current);
   }
   normaliseProbabilities(Current);
   return Current;
```

Algorithm 5-2: generatePerceptsAndProbs. P = percept, A = action, PSPOs = planning operator set.

Sections 5.4.1, 5.4.2 and 5.4.3 give examples of the application of these algorithms to the "slippery gripper" domain.

5.3.3 Filter by Precedence

Filtering of rules by precedence is an important algorithm for understanding the operation of the system. The algorithm removes all operators that defer to other operators in the P-SPO set and would not, therefore, have any effect on the output percept.

```
filterByPrecedence(PSPOs)
   orderByOutcomes(PSPOs);
   for (Pspo1 ∈ PSPOs){
      //start from next PSPO to avoid repetition
      for (Pspo2 = next(Pspo1); Pspo2 ∈ PSPOs) {
         //check for conflicts in outcomes
         if (conflicted(Pspo1.P_O ∪ Pspo2.P_O)) {
            //has precedence if not a member of "defers to" set
            if (Pspo2  ∉  Pspo1.P_D) {
               //check that precedence has been set
               //if precedence not set then remove 2 if 1 more general
               if ((Pspo1  ∈  Pspo2.P_D)
                  ||(|Pspo1.P_C|  <=  |Pspo2.P_C|)){
                  //remove conflicting operator
                  Pspo2 = prev(Pspo2);
                  remove (PSPOs, next(Pspo2));
               }
            } else {
               //iff operators have equal number of
               //outcomes then 2nd can supersede
               //(with check that precedence set)
               if ((Pspo1  ∈  Pspo2.P_D)
                  ||(|Pspo1.P_C|  <=  |Pspo2.P_C|)){
                  if (|Pspo1.P_O| == |Pspo2.P_O|) {
                     //remove and continue with next
                     Pspo1 = next(Pspo1);
                     Pspo2 = next(Pspo1);
                     remove (PSPOs, prev(Pspo1));
                  }
               }
            }
         } else {
            //if already checked conflicting outputs at this output
            //count then no further conflicts can occur at this level.
            Skip to the next output count (pseudocode omitted)
         }
      }
   }
```

Algorithm 5-3: filerByPrecedence. PSPOs = planning operator set.

The algorithm first orders the operators by outcomes. Recall from section 5.3.1 that all P-SPOs passed to *filerByPrecedence* have resolved variables (there are no free variables because all have been matched to the input percept).

Order by outcomes orders the operators: first by the number of perceptual features in the outcome set of the operator, and next by the unique identifiers of the ordered perceptual features within the outcome set. The result is an ordered list with operators with the greatest number of outcomes at the start of the list, and fewest operators at the end of the list. Outcome sets with greater numbers of outcomes always take precedence over those with fewer, enabling the filtering from greatest to least outcomes. Outcome sets with the same number of outcomes can be skipped if they do not cover the same outcomes.

The function *conflicted* is defined in the background knowledge and is used to determine whether the outcome sets of two operators are in conflict. If the union of the two outcome sets passed to *conflicts(P)* returns true, then the operators are in conflict.

If the operators are in conflict then one of them must be removed. In general, the rule with precedence will remove the rule that defers to it. In some cases, however, precedence will not have been set due to insufficient training data for P-SPO learning. If this is the case then it is still necessary to remove one of the conflicting operators. The most general operator (the operator with the highest support count or alternatively the least conditions) is kept because this is the one created from the greatest quantity of training data.

5.4 Successor Percept Generation Examples

The following sections show how successor percepts are generated from the rules and the current percept in particular situations.

5.4.1 Generation of a Successor Percept with one Applicable Operator

The simplest example of successor percept generation occurs when a single operator is applicable and has a single outcome. In this instance:

- The successor percept is initialised to be a copy of the current percept (implementing the frame rule because all successor percepts are unchanged).
- The percept element that conflicts with the outcome of the rule is replaced with the outcome percept element.

Example 1: Single Outcome

The *pickup(g,b)* action is applied to an initial percept:

$$painted(b, false),\ dry(g, true),\ holding(b, false), reward(none)$$

Examining the "slippery griper" operators we see that only one, rule (5) matches the input percept and action. The matched version of this operator is:

$$pickup(g,b) : dry(g, true), holding(b, false) \rightarrow \{1.0 : holding(b, true)\}$$

The outcome of this operator is *holding(b,true)*. If we apply the outcome to the initial percept the conflicting percept element is *holding(b,false)* (defined by the *conflict* function in the background knowledge). This element is removed and replaced by *holding(b,true).* In this instance there is only one outcome, so the sample output will be equivalent to generating all outputs. The changed output is in bold:

$$painted(b, false),\ dry(g, true),\ \mathbf{holding(b,true)}, reward(none)$$

An operator with more than one outcome can generate multiple possible successor percepts. A sample percept will produce a single successor percept, according to the probability of each outcome, while a full percept set will include all possible successor percepts and their associated probabilities.

Example 2: Multiple Outcomes

The *dryer(g)* action is applied to an initial percept given by:

$$painted(b, false),\ dry(g, false),\ holding(b, true), reward(none)$$

Examining the "slippery gripper" operators we see that only one, operator (4), matches the action *dryer* and conditions containing *dry(g, false)*.

$$dryer(X) : dry(X, false) \rightarrow \begin{Bmatrix} 0.9 : dry(X, true) \\ 0.1 : dry(X, false) \end{Bmatrix}$$

The variable *X* in the parameter is instantiated to *g* by the action, giving the outcomes *dry(g, true)* (probability 0.9) and *dry(g, false)* (probability 0.1). The successor percepts are generated by copying the initial percept and modifying the features in the outcomes that would cause a conflicted state.

Features are defined by the *conflict*ed function in the background knowledge, which in this case, indicates that the *dry(g, false)* literal in the initial state should be replaced. The successor percepts and probabilities are shown below. The *dry(g, false)* literal can be re-asserted without conflict.

$$painted(b,\ false), dry(g,\ false), holding(b, true), reward(none) \quad (pr:0.1)$$
$$painted(b,\ false), dry(g, true), holding(b, true), reward(none) \quad (pr:0.9)$$

Note: the first percept is unchanged from the initial percept. It would, therefore, be possible to re-write the P-SPO for the *dryer* action as:

$$dryer(X):dry(X,\ false) \rightarrow \begin{Bmatrix} 0.9:dry(X,true) \\ 0.1:no\ change \end{Bmatrix}$$

P-SPOs do not employ this form in the current research because:

(i) There is no restriction that perceptual features in the outcome set must be contained in the conditions of the operator. In the general case, this means that the operator cannot define a perceptual feature as unchanged because the initial value is not known. Examples of operators from the "slippery gripper" domain which do not contain the output feature in the conditions operators are: (7), (8), (9), (10) and (11) (see Figure 5.3).

(ii) Outcome sets must contain all values for any perceptual feature defined in any part of the outcome set for the *filterByPrecedence* algorithm to be well defined (discussed in section 5.3.3).

5.4.2 Generating Successor Percepts with Multiple Non-Conflicting Operators

Multiple P-SPOs can be applied in parallel to generate successor percepts with the restriction that only one operator can be applied to an individual perceptual feature. Conflicts are identified via the *conflicted(P)* function. A successor percept is generated by copying the initial percept and applying all operators matching the action and conditions.

Example

The *paint(g,b)* action is applied to the initial percept:

$$painted(b,\ false), dry(g, true), holding(b, true), reward(none)$$

Examining the "slippery gripper" domain operators we see that P-SPOs (1) and (3) apply to the percept and action combination.

$$paint(X,Y):holding(Y,true) \rightarrow \{1.0:painted(Y,true)\} \quad (1)$$
$$paint(X,Y):dry(X,true) \rightarrow \begin{Bmatrix} .4:dry(X,true) \\ .6:dry(X,\ false) \end{Bmatrix} \quad (3)$$

Operator (1) states that application of the *paint* action while holding the block results in the block being painted. Notice that we do not need the condition *painted(Y, false)* because the

block will always be painted in the successor state if the *paint* action is used when holding a block, irrespective of the initial painted condition.

Operator (3) states that the *paint* action with *dry(X, true)* results in *dry(X, false)* 60% of the time and *dry(X, true)* 40% of the time. These operators are not in conflict (they refer to different percept features). We therefore resolve the free variables to those in the current percept and apply both rules to find the successor state.

After application of operator (1) there is a single output percept (probability 1.0):

$$\textbf{\textit{painted(b,true)}},\ dry(g,true),\ holding(b,true),\ reward(none) \qquad (pr:\ 1.0)$$

Application of operator (3) to the percept generated by operator (1) gives two possible successor percepts:

$$painted(b,true),\ \textbf{\textit{dry(g, false)}},\ holding(b,true),reward(none) \qquad (pr:\ 0.6)$$
$$painted(b,true),\ \textbf{\textit{dry(g,true)}},\ holding(b,true),reward(none) \qquad (pr:\ 0.4)$$

A sample output will apply one of the outcomes of rule (3) according to the probability, producing a single successor percept.

Generation of all successor percepts will produce a list of successors and associated probabilities, calculated by multiplying the probabilities of each outcome in turn. The outcome set for operator (1), *painted(b, true)* given has a probability of 1.0. The probability of the output states is therefore equal to the probability of the outcome set of operator (3). The probabilities are 0.6, the probability of *dry(g, false)* given in operator (3), and 0.4, the probability of *dry(g, true)*.

5.4.3 Calculating Successor Percept Probabilities with P-SPOs

The probability of two independent events occurring simultaneously is calculated by finding the product of the probabilities of each independent event (see appendix section A.1). P-SPOs model dependent events using combined outcomes. Outcome sets can always, therefore, be treated as independent events to find the probabilities of successor percepts, with dependencies modelled within the outcome set.

Examining the "slipper gripper" domain operator set, it can be seen that the *paint(g,b)* action has multiple probabilistic outcomes. Take, for example, an initial percept given by:

$$painted(b,false),\ dry(g,true),\ holding(b,false),reward(none)$$

Operators (2) and (3) apply:

$$paint(X,Y): \begin{matrix} painted(Y, false), \\ holding(Y,true) \end{matrix} \rightarrow \left\{ \begin{matrix} 0.1: painted(Y,true) \\ 0.9: painted(Y, false) \end{matrix} \right\} \quad (2)$$

$$paint(X,Y): dry(X,true) \rightarrow \left\{ \begin{matrix} 0.4: dry(X,true) \\ 0.6: dry(X, false) \end{matrix} \right\} \quad (3)$$

When resolved, operator (2) states that using the *paint(g,b)* action with the percept elements *holding(b, false)* and *painted(b, false)* results in *painted(b, true)* with probability 0.1 and *painted(b, false)* with probability 0.9. Applying operator (2) outputs two partial successor percepts and associated probabilities, which become the inputs for the next operator.

painted(b, false), *dry(g,true), holding(b, false), reward(none)* *(pr : 0.9)*
painted(b,true), *dry(g,true), holding(b, false), reward(none)* *(pr : 0.1)*

When resolved, operator (3) states that using the *paint* action with *dry(g, true)* results in *dry(g, false)* with probability 0.6, and *dry(g, true)* with probability 0.4. If we apply this rule to the first percept above, the conflicting element is *dry(g, true).* The output percepts are constructed by removing the *dry(g, true)* element and replacing it with the outcomes for rule (3). The probability is the product of the probability of the partial percept, 0.9, and the probabilities of the outcomes of rule (3) giving the percepts below. Notice that the probabilities sum to give 0.9, the probability of the first of the partial percepts above.

painted(b, false), ***dry(g, false)***, *holding(b, false), reward(none)* *(pr : 0.54)*
painted(b, false), ***dry(g,true)***, *holding(b, false), reward(none)* *(pr : 0.36)*

Similarly, if we apply operator (3) to the second partial percept the output percepts are:

painted(b,true), ***dry(g, false)***, *holding(b, false), reward(none)* *(pr : 0.06)*
painted(b,true), ***dry(g,true)***, *holding(b, false), reward(none)* *(pr : 0.04)*

There are no further matching operators to apply. Our possible successor percepts are therefore the four combined operators above. The sum of all possible successor percept probabilities is always 1.0.

5.4.4 Conflicting Operator Outcomes

Conflicting operator outcomes occur when more than one outcome refers to the same perceptual feature, which will happen regularly in complex environments. An example based on conflicts in operators learned from data will be used here to demonstrate the concept.

The planning operators used in this research are designed to be: (i) machine learnable; (ii) applicable across novel environments beyond the initial training set. Operators with multiple conditions (specific) are built from those with fewer conditions (general). The general

operators are kept when a more specific one is created because the general operator may apply to novel situations which were not part of the training set. This can only occur in percepts with non-Boolean features, because an additional condition would only be learned if it added information. The operator learning algorithm can be tasked to learn the *conflicted* function (given as background knowledge in this chapter) with the result that it may be incomplete and the model will not necessarily define all values of a perceptual feature if they have not been encountered.

Recall the *paint* action's effect on the *painted* perceptual element defined by operators (1) and (2) from the "slipper gripper" domain operator set:

$$paint(X,Y): holding(Y,true) \rightarrow \{1.0: painted(Y,true)\} \quad (1)$$

$$paint(X,Y): \begin{matrix} painted(Y,false), \\ holding(Y,false) \end{matrix} \rightarrow \left\{ \begin{matrix} 0.1: painted(Y,true) \\ 0.9: painted(Y,false) \end{matrix} \right\} \quad (2)$$

While learning the paint action described by the operators, the operator learning algorithm would also derive an operator with fewer conditions, such as:

$$paint(X,Y): painted(Y,false) \rightarrow \left\{ \begin{matrix} 0.3: painted(Y,true) \\ 0.7: painted(Y,false) \end{matrix} \right\}$$

This operator describes the result of the paint action on an unpainted block over the available evidence, irrespective of whether the block is held in the gripper. This operator should be kept in the final rule set in-case we have not seen all possible values the *holding* perceptual feature. For example, *holding(X, Y)* may have three values for *Y* rather than the two we have observed: *holding(X, true)*, *holding(X, false)* and *holding(X, partial)* (to indicate that the block is partially held). If we only retained operators with *holding(X, true)*, or *holding(X, false)* there would be no applicable rule for this previously unseen situation.

Both the more specific and less specific versions of this operator are applicable to any state with matching conditions, but only one may be applied to produce successor percepts because they apply to the same perceptual feature.

The conflict resolution strategy employed is to give operators an order of supremacy over each other. This is established empirically by giving precedence to the operator that provides the most accurate probability distribution over outcomes for the combined conditions. This method will tend to give supremacy to the more specific rule. See section 6.6 for a definition of the *precedence* algorithm.

5.4.5 Remove Invalid States

The final step in the percept generation algorithms presented in 5.3 is to remove invalid states. These can be generated if the model of the environment is incomplete.

The learned operator set can be incomplete due to insufficient training data or restrictions on the operator outcome size during learning (causing dependencies in the outcome sets to be omitted). If the P-SPO set it incomplete, background knowledge can be employed to identify generated percepts that are in an invalid state. The state generation function resolves this issue by removing invalid percepts from the output set and normalising the remaining outcomes such that the combined probability sums to 1.0.

An example of a constraint from the two block world is that it is not possible to be holding block *b1* and holding block *b2* simultaneously. The operator set does not state this, and it may be that an incomplete operator set, such as this, is generated from the available evidence (although it is easy to see that this is not a deficiency of the syntax because we could add an extra condition to *pickup* to indicate that the gripper must not already be holding a block). This is distinct from the use of *conflicted(P)* in the background knowledge, which tells us that individual perceptual features cannot be in a conflicted state.

The *invalid* function can be defined for the *slipper gripper* scenario such that:

$$invalid(P) \leftarrow holding(X, true) \in P, holding(Y, true) \in P, X \neq Y.$$

Applying the pickup action to the two block scenario gives an example of this function. If the initial percept is given by:

$$painted(b1, true), holding(b1, true), painted(b2, false), holding(b2, false),$$
$$dry(g, false), reward(none)$$

The resolved pickup operator matching these conditions for *b2* is:

$$pickup(g, b2) : dry(g, false), holding(b2, false) \rightarrow \left\{ \begin{matrix} 0.6 : holding(b2, true) \\ 0.4 : holding(b2, false) \end{matrix} \right\}$$

This would produce the successor percepts:

$$\begin{matrix} \textbf{\textit{painted(b1, true), holding(b1, true), painted(b2, false),}} \\ \textbf{\textit{dry(g, false), reward(none), holding(b2, true),}} \end{matrix} \quad \textbf{\textit{(pr : 0.6)}}$$

$$\begin{matrix} painted(b1, true), holding(b1, true), painted(b2, false), \\ dry(g, false), reward(none), holding(b2, false), \end{matrix} \quad (pr : 0.4)$$

The percept in bold is removed because it is invalid, leaving the second percept, which has probability 0.4. Probabilities are normalised by dividing the probability of each valid percept by the total probability of the valid percepts. In this case, there is only one valid percept, giving 0.4/0.4 = 1.0.

5.5 Frame Assumption

The frame problem, first identified in logic-based planning by McCarthy & Hayes [57], is the problem of expressing the dynamics of a system without having to expressly state every aspect of the environment that is not affected by an action. The term derives from a technique in cartoon animation in which a static image (the frame) depicting the background of a scene is superimposed with the animated aspects of the scene.

The frame problem in logic is that specifying the conditions changed by an action does not allow you to conclude that all other aspects of the environment are unchanged. An obvious solution is to provide a rule for each action that states, for example, that if a *pickup* action is used on a block it does not change whether the block is painted. The number of these frame axioms is equal to the product of the number of features of the environment and the number of actions available. The problem with this solution is that each of these must be asserted at every time step, taking up a large amount of unnecessary processor time.

The solution proposed by Fikes & Nilsson [31] is to move the generation of future world states outside the standard logic using "extra-logical systems". Essentially this allows a future world state to be a copy of the current state and anything that is changed is removed or added by the operators. They use the term STRIPS for their operators in reference to the comic strip animation (or cartoon) basis of the frame problem with the solution being to remove the changed feature and add the new feature. The frame assumption used here employs the same solution method, by copying the current percept and then replacing any changed features using the outcomes of the operators.

The syntax of parallel stochastic planning operators allows the inclusion of operators with outputs but with no action and no conditions. Indeed, these are always present in a rule set learned from experience by ASDD (section 5.8) because they form the building blocks of later rules, and can be used as a default if no rule is applicable. These operators always match the current environment and we therefore need a method of preserving the compact modelling power of the frame rule, while allowing the flexibility afforded by operators with empty actions or conditions. This is achieved through relaxing the frame assumption to allow a special *environment* operator type, which often defers to an action if the action affects the same output variable. Environment operators have great expressive power and can model both

static environments (those that stay the same in the absence of agent action) and non-static environments (those that change irrespective of agent action).

The revised frame assumption is given below. Note that environment operators do not model the complete mechanics of the agent's environment, but rather, model the evolution of the agent's perception of its environment in the absence of action, or when its action does not affect particular environment features.

- **P-SPO Frame Assumption:** elements of the agent's successor percept will remain unchanged from the current percept if they are not present in the outcomes of any operator matching the selected action and current percept, and they are not present in the outcome set of any environment operator matching the current percept.

Several other solutions within the standard logic framework have been proposed. The most relevant is the successor-state axiom solution proposed by Reiter [56][75]. This states that an environment feature will be true after the execution of an action if and only if:

(i) the action causes the environment feature to be true; or

(ii) the environment feature was already true and the action does not cause it to become false.

5.6 Pure Environment Actions

The syntax for P-SPOs allows the action to be empty (represented by {}). This is referred to as a *pure* environment action, because it defines the effects of the environment on perceptual features that are not part of the outcomes of any action. These are environment features that are entirely beyond the agent's control and can be used to model, for example, the weather or random noise events.

An example in the "slippery gripper" domain is to add a new *weather(X)*, environmental feature, where $X \in \{sunny, cloudy, raining\}$. The environment variable could be modelled by the additional operators.

$$\{\} : weather(cloudy) \rightarrow \left\{ \begin{array}{l} .3 : weather(sunny) \\ .4 : weather(cloudy) \\ .3 : weather(raining) \end{array} \right\}$$

$$\{\} : weather(raining) \rightarrow \left\{ \begin{array}{l} .7 : weather(cloudy) \\ .3 : weather(raining) \end{array} \right\}$$

$$\{\} : weather(sunny) \rightarrow \left\{ \begin{array}{l} .5 : weather(sunny) \\ .5 : weather(cloudy) \end{array} \right\}$$

Operators with an empty action can only be used to model situations in which there is no agent action effect on the environment feature. If this was not the case, the pure operators would interfere with the frame assumption.

5.7 Environment Operators

Environment operators are special case rules that are used to model environment features that can be affected by agent actions, but for which the currently selected action has no effect. The empty action condition cannot be used for this because it matches all actions and will contain the probabilities associated with the action as well as the probabilities when the action is not taken.

If, for example, we wished to model the environment action that a wet dripper may become dry with probability .05 each time step irrespective of the action taken, it would be tempting to model this as:

$$\{\} : dry(X, false) \rightarrow \left\{ \begin{array}{l} .05 : dry(X, true) \\ .95 : dry(X, false) \end{array} \right\}$$

The empty action, however, matches all the actions in the environment, and would therefore also match situations that change the state of the *dry* feature, such as the *dry* and *paint* actions. This would give us probabilities based on the combined situations in which *dry(X, false)* was present in the percept.

The *environment* operator can be used to overcome this. The operator tells us what happens when the currently selected action does not affect the given feature. The syntax is:

$$environment : dry(X, false) \rightarrow \left\{ \begin{array}{l} .05 : dry(X, true) \\ .95 : dry(X, false) \end{array} \right\}$$

This is a powerful feature of the syntax, because it allows the compact representation of features that are not affected by domain operators, even if they are subject to change. The frame assumption does not provide this flexibility, only allowing us to model features that remain unchanged.

Figure 5.3 shows that the initial set of P-SPOs for the "slipper gripper" domain contain the operator:

$$environment : \{\} \rightarrow \{1.0 : reward(none)\}$$

This operator allows us to succinctly model that no reward is given unless the *new* operator is called. If we did not have recourse to environment operators, a separate rule would have to be given for each operator to state this fact, because the environment feature does not remain unchanged.

If, for example, the *new* action was selected in the previous time step and a positive reward was received, the selection of the *dry* action would, in effect, change the state of the reward feature from *reward(positive)* to *reward(none)*.

When using P-SPOs learned from data, it is necessary to use the *environment* operator in all situations to replace the frame assumption because the learned rule set will contain partial operators which would produce incorrect output.

Take for example, the operator:

$$dryer(X) : dry(X, false) \rightarrow \begin{Bmatrix} 0.9 : dry(X, true) \\ 0.1 : dry(X, false) \end{Bmatrix}$$

When learning this operator, the rule learning algorithm will develop partial rules, such as:

$$dryer(X) : \{\} \rightarrow \begin{Bmatrix} 0.95 : dry(X, true) \\ 0.05 : dry(X, false) \end{Bmatrix}$$

This rule has no conditions and its outcome probabilities are, therefore, built from a combination of all observations of the *dry* outcome for the *dryer* action, including the operator which is implied by the frame assumption:

$$dryer(X) : dry(X, true) \rightarrow \{1.0 : dry(X, true)\}$$

The environment operator for this is:

$$environment : dry(X, true) \rightarrow \{1.0 : dry(X, true)\}$$

The environment operator can be given precedence over the partial *dryer* operator above, if its outcome probabilities are a more accurate representation of the data. The frame assumption defines what happens in the absence of an operator, and thus cannot be given precedence. Environment operators represent a relatively compact solution in dynamic environments, because we only need one operator for each perceptual feature, rather than one for each action and feature combination.

5.8 Summary

This chapter has defined the syntax of parallel stochastic planning operators and algorithms for successor percept generation using them. The operators have advantages over other stochastic rule representations in dynamic environments in that they can model both an environment and the action of an agent within the environment using a minimal set of operators. The compact representation is important if the operators are to be acquired from data because all available evidence can be used in evaluating probabilities. The representation is also useful in splitting the state-space the operators represent into meaningful sections for the RVRL algorithm (presented in chapter 9).

6. Learning Parallel Stochastic Planning Operators

This chapter presents the Apriori Stochastic Dependency Detection (ASDD) algorithm and an optimised variant (ASDDs). ASDD is an efficient algorithm for constructing parallel stochastic planning operators (P-SPOs) from observation data using a combination of statistical significance and association-rule mining methods. The algorithm and its variant are novel contributions of this research. ASDD was first presented in [18]. ASDDs is presented here for the first time.

ASDD uses a fast association rule mining method, based on the Apriori algorithm (defined by Agrawal and Srikant [1]) to generate candidate rules, and then filters the generated rules using statistical significance to generate a minimal rule-set.

The process of learning P-SPOs from data consists of the following five steps:

1) *Find rule sets:* find common occurrences of *action* and *context* leading to *outcome* in a set of perceptual data gathered from experience in an environment.

2) *Establish empirical probability:* find the empirical probability of the outcome if the action and context are observed.

3) *Filter the generated rules to remove conditions that are not statistically significant:*

   (i) Filter candidate rules as they are generated, avoiding the generation of redundant candidates.

   (ii) Filter after completion of the candidate generation process to remove rules.

4) *Combine rule sets to form P-SPOs:* combine rule sets with the same context which refer to the same perceptual feature(s) to form P-SPOs.

5) *Establish P-SPO precedence:* establish precedence between operators for conflict resolution.

The P-SPO set is used as an environment model in the framework presented in chapter 4, and the process of learning the operators is, therefore, a model learning process. As discussed in section 3.1, a simple model can be learned by keeping a record of the successor states that follow from a state and action, or, in the case of a situated agent, the successor percept following a percept and action. The process of P-SPO learning takes this concept as a starting point, and builds the model by finding commonly occurring sets of features within the percepts.

For example, an agent in the slippery gripper domain (defined in chapter 5) can build a set of perceptual data items by taking random actions in an environment. The perceptual data items contain:

- The percept before the action was taken.
- The action taken.
- The percept received after the action was taken.

Table 6-1 gives a subset of an agent's perceptual data items is the slippery gripper domain. The process of learning P-SPOs involves finding actions and context (elements of the percept) that commonly occur with elements of the successor percept. In this case, there is a set of commonly occurring elements (highlighted in bold text) indicating that taking the action *dryer(g)*, in the context that the gripper was previously wet, *dry(g, false)*, leads to the successor percept *dry(g, true)*. In one instance, the successor contains, *dry(g,false)*. This follows from the fact that the *dryer* action in the slippery gripper domain is stochastic and will sometimes fail. The algorithm must be able to cope with these stochastic outcomes.

Table 6-1: A sample of perceptual data items for the "slippery gripper" block painting agent.

Percept	Action	Successor Percept
…	…	…
***dry(g, false)**, holding(b, true), painted(b, false), reward(none)*	***dry(g)***	***dry(g, false)**, holding(b, true), painted(b, false), reward(none)*
dry(g, false), holding(b, true), painted(b, false), reward(none)	*paint(b)*	*dry(g, false), holding(b, true), painted(b, false), reward(none)*
***dry(g, false)**, holding(b, true), painted(b, true), reward(none)*	***dry(g)***	***dry(g, true)**, holding(b, true), painted(b, true), reward(none)*
…	…	…
***dry(g, false)**, holding(b, false), painted(b, true), reward(none)*	***dry(g)***	***dry(g, true)**, holding(b, true), painted(b, true), reward(none)*
…		
***dry(g, false)**, holding(b, true), painted(b, true), reward(none)*	***dry(g)***	***dry(g, true)**, holding(b, true), painted(b, true), reward(none)*
…	…	…

Based on the evidence in the sample data, the rule probability of the outcome can be found empirically by finding the number of times the outcome occurs following the context and the action, and the number of times the outcome did not occur following the context and the action. If all of the examples of the pattern are present in the above table, we have:

- *dry(g,false)* follows from observing *dry(g,false)* and taking action *dryer(g)* in 1 out of the 4 examples, giving probability 0.25.

- *dry(g,true)* follows from observing *dry(g,false)* and taking action *dryer(g)* in 3 out of the 4 examples, giving probability 0.75.

Combining these partial rules, a P-SPO can be created:

$$dryer(g) : dry(g, false) \rightarrow \begin{Bmatrix} 0.25 : dry(g, true) \\ 0.75 : dry(g, false) \end{Bmatrix}$$

The following sections:

- Present the ASDD algorithm for rule set discovery.
- Present the supplementary algorithms needed to combine rule sets in P-SPOs and to establish supremacy between P-SPOs
- Present a variant of the algorithm for fast set searching.

The slipper gripper domain will be used to demonstrate the rule learning concepts.

6.1 A Note on Learning Planning Operators from Experience

The majority of work in the planning community has centred on search optimisation, via techniques such as constraint satisfaction [3][78]. These techniques assume that a human designer is able to provide the required planning operators. Often, the design of these operators is difficult because the mechanics of the agent's environment are poorly understood, or in the case of stochastic environments, the random elements are difficult to model. Stochastic planning operators can, however, provide a useful level of generalisation for an agent in a deterministic environment, in addition to modelling environments that are inherently random.

Empirical learning of parallel stochastic planning operators is challenging because:

- An action may have uncertain outcomes inherently.
- The outcomes of an action may be masked by external interference.
- The action conditions may be masked by external elements.

The key point is that the planning operator structure must be one that can be learned through empirical data. The P-SPOs defined are learnable because they do not rely on hidden variables, or random variables in the conditions set (such as those found in Poole's independent choice logic [71]). The rule structure allows the agent to learn both the outcomes of its actions, and the influence of the environment (which is beyond its control).

The ability to model independent outcomes using parallel operators is useful when operators are acquired from experience because it allows relevant evidence to influence the estimate of the probability of the operator output.

6.2 Learning P-SPOs with ASDD

ASDD is an algorithm for learning parallel stochastic planning operators, based on the Apriori algorithm for mining association rules [1], and the Multi-Stream Dependency Detection (MSDD) algorithm for finding dependencies in multiple streams of data [64]. The algorithm is one of the main contributions of this research and was presented previously in [18]. In previous research by the author, MSDD has been applied to the task of learning probabilistic planning operators with a similar syntax to P-SPOs [17]. Probabilistic STRIPS operators have been learned using MSDD by Oates and Cohen [65].

6.3 Assumptions

Several simplifying assumptions are made to operator learning using the ASDD algorithm:

- *Operators are acquired from batch training data:* training data will be presented to the ASDD algorithm in batch form (extensions for in-line operator acquisition are discussed in section 11.3.3).
- *Operators are learned in ground form:* ASDD learns the P-SPOs from ground example data, and outputs a set of operators in ground form (extensions to the algorithm to enable the substitution of variables are discussed in section 11.3.4).
- *Conditional independence:* The outcomes of actions and environment operators to be modelled by ASDD are conditionally independent. The current implementation of ASDD therefore learns a single output for each operator (extensions to the algorithm to learn non-independent outputs are discussed in section 11.3.5).

Conditional independence is a strong assumption that does not hold for some of the examples used in this thesis. Issues created by this are resolved, to some extent, by the use of the *invalid(P)* function defined in background knowledge (section 5.4.5).

6.4 ASDD: Apriori Stochastic Dependency Detection

The language used to describe ASDD reflects that used in [1]. The main algorithm is similar, with an additional *aprioriFilter* step which removes potential conditions from rules if they are shown to have no significant effect on their probability. There is also a final *filter* step, which is equivalent to that used in MSDD and removes conditions at a higher level of significance to

produce a final operator set. Apriori uses *support* (a measure of the percentage of the data which contains a rule), while ASDD uses *support count*, which is a count of the number of occurrences of a rule *(*section 6.4.2).

At the highest level the tasks of the ASDD algorithm are to:

1) Efficiently generate candidate operators which may be significant to the outcome set.
2) Remove operators which are not significant to the outcome set.

Supplemental algorithms learn specific P-SPO based elements, including:

- *CreateP-SPOs:* The ASDD algorithm creates an individual rule for each outcome set. *P-SPOs* are created by combining these individual rules (section 6.4.12).
- *AddMissing:* an algorithm to add missing parts of an operator's outcome set. A P-SPO with an outcome set with probabilities which do not sum to 1.0 indicates a rule was missed which should have formed part of a P-SPO.
- *Precedence:* an algorithm for defining precedence between P-SPOs. Operator precedence defines which rule should be applied when conflicting outcomes occur.
- *Frame and Environment Operators:* perceptual features in the outcome set which are unaffected by actions must be identified by the algorithm. If the feature retains its value in all circumstances unless affected by an action it is captured by a set of *frame rules*. If a feature changes independently of actions, this is captured by *environment* operators.

Association rules generated by algorithms such as Apriori have precedence based on a *confidence* measure. *Confidence* indicates the probability of the rule's outcome, given its conditions. Outcome sets in P-SPOs have an outcome probability which is generated empirically in the same way as *confidence*, but the rules need a further measure of the validity of the rule. This is achieved by first filtering rules that have unnecessary complexity, using the *aprioriFilter* function (section 6.4.9). Once a minimal rule set has been established, the *precedence* algorithm establishes supremacy between rules in conflict situations (section 6.6).

The task of learning P-SPOs from data (in the ground case) is that of finding significant associations between sets of perceptual features and actions at time *t-1*, and perceptual features at time *t*.

6.4.1 Convert Sensor Data Percept to Perceptual Feature Axioms

The agent body's sensors are updated each time step as a function of the current state of the environment. This sensor data is in a raw form which is not specific to the type of agent that is interpreting it. The agent's *perceive* function maps sensor data to a percept, and can be defined in any appropriate way for the intended modelling mechanism. In this case, raw sensor data must be converted into a set of perceptual feature axioms in order to enable P-SPO learning. The definition of the *perceive* function is domain dependent (see chapter 7 for examples) and the following section are therefore presented using the post-processed percept in perceptual feature axiom form.

6.4.2 Perceptual Data Items (PDIs)

A percept, P, is a set of perceptual feature axioms. A perceptual data item (PDI) contains the percept received and action taken at time $t\text{-}1$, and percept received at time t.

The PDI data set, D is defined as a set of perceptual data items (*PDIs*) for an agent where each *PDI* is a triplet of the form $\{P_{t-1}, A_{t-1}, P_t\}$.

The PDI received at time t is defined as:

$$d_t = \{p_{t-1}, a_{t-1}, p_t\} \tag{6.1}$$

6.4.3 Rule Element Sets

The possible elements of $P \cup A$ are collectively known as *rule elements*. A *PDI* contains the rule element set x, if $x \subseteq p_{t-1} \cup a_{t-1} \cup o$, where o is an *outcome* rule element, and $o \in p_t$.

The rule element set, x, has *support count, sc,* in the perceptual data item set D if sc of the PDIs in D contain x.

The constant *MINSUP* defines the minimum support count a rule element set must display before it is admissible as a candidate for the next iteration of the algorithm, or to the rule base. The *support count* in ASDD replaces *support* in the Apriori algorithm. *Support* is defined as a percentage of the PDIs in D containing x, rather than a count. The change to use *support count* in ASDD is made to aid discovery of rare outcomes which may be statistically significant.

Two additional features are present in size-one rule element sets:

- *id*: an additional integer identifier field, used to speed up comparison between individual rule elements identified in the data set. If the identifier is stored as a string, the *id* can be calculated as a hash function on the string.

- *pfs*: perceptual feature set. The set of "one rule element" perceptual features which this rule element belongs to. Size-one rule element sets are grouped into perpetual feature sets. These sets are equal in all but the last element and can be identified from data (see section 6.4.11). For example, the rule elements *dry(g, false)* and *dry(g, true)* are only different in the last element and therefore belong to the same perceptual feature set.

6.4.4 Rules

A *rule* is a rule element set which contains an outcome.

Using syntax similar to that used in stochastic logic programming [61]:

- The *head* of the rule is the outcome, *o*.

- The *body* of the rule is the remaining rule elements (the conditions).

- The *probability* of the head occurring given that the body is observed, *pr*.

The probability can be calculated empirically as the number of PDIs in the data set, *D*, containing the *head* and *body,* divided by the number of PDIs that contain the *body*. In other words, the number of times the outcome follows the conditions of the rule.

6.4.5 Rule Set Discovery

The problem of discovering a rule set can be separated into four sub-problems:

1) *Discover regularly occurring rule element sets:* discover sets at level *k* exhibiting support count*, sc,* above *MINSUP*. The level of a rule element set is defined as the number of rule elements it contains (section 6.4.6).

2) *Combine rule element sets:* rule elements at level *k* are combined to form a list of candidate sets for level *k+1* using *aprioriGen*, which removes all candidates that cannot have minimum support (section 6.4.8).

3) *AprioriFilter*: after level 3, the *AprioriFilter* filter function is applied to remove candidate rules (rule element sets containing an outcome element) at level *k*, which are *covered* by an operator at level *k-3* (section 6.4.9).

4) *Filter*: Finally, the *filter* function is applied to the remaining rules to remove rules which are *covered* by a rule at any level (section 3.1.3).

Sub-problems (1) and (2) are as defined by the Apriori algorithm with a change to use *support count* in place of *support*. Sub-problem (3) is a new addition for ASDD. Sub-problem (4) is a new addition using a function defined in MSDD [64].

The notation used in ASDD and the associated algorithms is:

- *L[k]*: the set of rule element sets of size *k* which display minimum support. Each member of this set has four fields:
  - (i) *x:* a set of rule elements.
  - (ii) *sc:* support count (number of times the rule elements *x*, matched the database). If *x* does not contain an outcome, *o*, this will be equal to *bs* (below).
  - (iii) *bs:* the support count of the body (the rule element set, *x,* excluding the outcome*, o)*, of the rule (the number of times the body of the rule matched the database).
  - (iv) *pSet:* a set of references to other rules with the same conditions and matching output perceptual feature(s), which will be combined to form a P-SPO.
- *C[k]*: the set of candidate rule element sets of size *k* (sets with potentially large support count). Fields are identical to *L[k]*.

The empirical probability, *pr*, of a rule is a function returning *sc* divided by *bs*.

6.4.6 Discovering Regularly Occurring Rule Element Sets

Discovering regularly occurring rule element sets using the Apriori method involves making multiple passes over the perceptual data set *D*. In the first pass (level $k = 1$) the support of each rule element set of size one is counted to determine which of them occurs regularly enough in the data to be included (i.e. has minimum support). In each subsequent pass, regularly occurring rule element sets from the previous pass (level $k-1$) are used to create *candidate* rule element sets.

The support for each of these candidate sets is counted in a pass over the data. Candidates that do not have minimum support are removed and the remaining candidates are used to generate candidates for the next level.

After the third pass, rule element sets that have an outcome element (rule head) can be *filtered* by rules at the *k-3rd* level with the same outcome, in order to evaluate the additional conditions. If the additional conditions do not have a significant effect on the probability of the outcome (section 6.4.9) they are discarded. This process continues until no new sets of rule elements are found.

The *AprioriGen* algorithm (adapted from [1]) generates the candidate rule element sets to be counted in a pass by combing the rule element sets with high support count in the previous pass. Candidates with *k* rule elements are generated by combining rule element sets at the *k-1* level. Any generated candidates at level *k* containing a subset at level *k-1* which does not have minimum support are then removed in the prune step, because any subset of a large set must also be large. Each candidate removed by this step avoids the need for an expensive pass over the data set when generating candidates.

6.4.7 The ASDD algorithm

```
ASDD(D)
   L[1] = extractOneRuleElementSets(D);
   for (k = 2; L[k-1] ≠{}; k++){
      C_K = aprioriGen(L[k-1]);            //(step 1)
      for all (pdi ∈ D){                   //(step 2)
         C_t = subset(C_K, pdi)
         for all (c ∈ Ct)
            c.sc ++;
      }
      L[k] = {c ∈ C_K | c.sc ≥ MINSUP}     //(step 3)
      if (k > 3)                           //(step 4)
         L[k] = aprioriFilter(L[k], L[k-3], APRIORI_G);
   }
   ruleSet = {};
   for (k = 1; L[k] ≠{}; k++)
      ruleSet = ruleSet ∪L[k];
   return filter(ruleSet, FINAL_G);
```

Algorithm 6-1: ASDD. D = database of perceptual data items

The first line of the ASDD algorithm counts the occurrences of single rule elements in D to determine one-rule element-sets that have a high support count (this step has been altered slightly from that used in Apriori to extract perceptual feature information from the data). Each repeat of the loop consists of the following four steps:

1) Large rule element sets *L[k-1]* found in the pass (*k-1*) are used to generate the candidate rule element sets *C[k]*, using the *aprioriGen* function (section 6.4.8).

2) The *support count* of candidates in *C[k]* is determined by performing a database scan using the *subset* function, which returns the subset of the candidates, *C[k],* contained in each PDI.

3) Rule element sets with below minimum support are removed.

4) Rules (rule element sets containing an outcome element) are filtered against rules that subsume them at the level *k-3* by the *aprioriFilter* function (section 6.4.9).

The loop is repeated until: (i) no further candidates are generated; (ii) a maximum level has been reached (resulting in rules with a maximum number of conditions); or (iii) a maximum number of rules has been generated.

Finally, rules at all levels are combined into a single list (sorted by generality) and are tested for statistical significance by the *filter* function. The *filter* function tests for significance at a higher significance than *aprioriFilter,* and tests for rules that are covered at all previous levels.

The slowest part of the algorithm is the loop for each PDI around the *subset* function. Section 6.6 shows an algorithm for increasing the speed of this function which has a trade-off of requiring additional storage.

The initial implementation of *subset* cycles through all candidate rule sets, and tests to see if the rule elements are contained in the PDI (the implementation used for the standard Apriori algorithm).

```
subset(C, PDI)
   subsetC = {};
   for all (c ∈ C)
      if (c.x ⊆ PDI))
         subsetC.add(c);
   return subsetC;
```

Algorithm 6-2: subset. C = candidate rule element sets. PDI = perceptual data item.

6.4.8 The aprioriGen Function

The *aprioriGen* function generates a set of potentially large rule element sets of size *k* from rule element sets of size *k-1*.

```
aprioriGen(Lk-1)
   Ck = join(Lk-1);
   return aprioriPrune(Ck, Lk-1);
```

Algorithm 6-3: aprioriGen. Lk-1 = candidates at level k-1

There are two steps, taken directly from the Apriori algorithm:

1) *Join: L[k-1]* rule element sets are combined with other *L[k-1]* rule element sets to form candidate rule sets *C[k]*. Join uses unique ids in the rule elements to avoid repeated candidates.

2) *Apriori-Prune:* generated candidates for which a rule element subset of size *k-1* is not present in *L[k-1]* are deleted.

ASDD adds the following steps to the *join* function:

1) *Restrict to single outcome*: outcomes are restricted to a single perceptual feature (see assumptions, section 6.3). If both *L[k-1]* rules have an outcome rule element (rule head) they are not combined.

2) *Restrict to probability < 1 parents*: parents with an outcome (rule head) and a rule probability of 1 are restricted from producing children because they already identify a definite outcome (no further improvement can be made to the rule).

3) *Copy body support count*: rules (rule element sets containing an outcome rule element) will have a body that is equal to one of the rules element sets that is used to form them. In this case, *bs*, the body support count is copied from the support count, *sc*, of previous rule element set in order to restrict the number of database passes required.

```
join(L)
   C = {};
   for all (p ∈ L) {
      //do not generate if p is a rule with prob=1
      if (head(p)≠{} and p.bs == p.sc)
         next p
      for all (q ∈ L where p ≠ q) {
         //do not generate if q is a rule with prob=1
         if (head(q)≠{} and q.bs == q.sc)
            next q
         //ignore if both have an outcome
         if (head(p)≠{} and head(q)≠{})
            next q;
         //only generate if last element id of p.x > q.x
         if (last(p.x).id > last(q.x).id){
            next q;
         }
         //only generate if all elements equal except last
         for (i = 0; i < |p.x|-1; i++) {
            if (p.x[i] ≠ q.x[i]) {
               next q;
         }

         //combine elements (last is different)
         newC.x = add(p.x, last(q.x));
         if (head(newC)≠{}){
            if (body(newC) == p.x)
               newC.bs = p.sc;
            else if (body(newC) == q.x)
               newC.bs = q.sc;
         }
         add(C, newC);
   }
   return C;
```

Algorithm 6-4: join. L = rule element sets at previous level.

Note: The function *body* returns all rule elements excluding outcome elements (rule head).

The *aprioriPrune* algorithm (as defined in [1]) removes candidates from the newly generated set by checking whether all sub-sets of the candidates have minimum support count. If a subset exists that does not have minimum support count, then the candidate cannot have minimum support count.

```
aprioriPrune(Ck, Lk-1)
   for all (c ∈ Ck)
      forall (k-1 size subsets s of c)
         if (s ∉ Lk-1)
            delete c from Ck;
```

Algorithm 6-5: aprioriPrune. Ck = candidates at level k. Lk-1 = rule element sets at level k.

Example of aprioriGen candidate rule element set generation

The example shows the generation of level 3 rule element sets from level 2 sets. A subset of the level 2 rule element sets for the "slippery gripper" painting robot domain are given below.

The subset shown is the rule elements are those related to the *dryer* action (those that will contribute towards building the full rules). Rule element sets can be created in any order, with the result that the outcome rule element may be positioned at any point in the set. Outcome elements are indicated by the $\rightarrow$ symbol, showing that this is an implication to a rule head. Probabilities can be generated for any rules with a rule head, but these are not relevant to the algorithm and are therefore not included until the filter step.

The level 2 rule element sets are:

(1) $\{dryer(g), dry(g, false)\}$
(2) $\{dryer(g), \rightarrow dry(g, true)\}$
(3) $\{dryer(g), \rightarrow dry(g, false)\}$
(4) $\{dry(g, false), \rightarrow dry(g, true)\}$
(5) $\{dry(g, false), \rightarrow dry(g, false)\}$
(6) $\{dry(g, true), \rightarrow dry(g, true)\}$
(7) $\{dry(g, true), \rightarrow dry(g, false)\}$
(8) $\{dryer(g), dry(g, true)\}$

Figure 6.1: Subset of the level 2 rule element sets related to the dryer action in the "slippery gripper" domain

Join: the join step creates the level 3 candidate rule element sets from the level 2 set by combining pairs of rule element sets if they meet the conditions:

(i) The two rule element sets have a matching first element and a non-matching 2nd element (in the general case, the algorithm will combine rules that are matching in all but the final element).

(ii) Only one of the pairs to be combined has an outcome.

All level 2 candidates have outcomes except rule element sets (1) and (8). The application of condition (ii) has the result that rules (1) and (8) are the only rules that can be combined with rules (2) through (7) to create level 3 rules.

Rule element set (1) can be combined with other rule element sets with the same first element, giving:

from (1) & (2) $\{dryer(g), dry(g, false), \rightarrow dry(g, true)\}$
from (1) & (3) $\{dryer(g), dry(g, false), \rightarrow dry(g, false)\}$

Rule element set 8 can be combined with other rule element sets with the same first element, giving:

from (1) & (8) $\{dryer(g), dry(g, false), dry(g, true)\}$
from (2) & (8) $\{dryer(g), dry(g, true), \rightarrow dry(g, true)\}$
from (3) & (8) $\{dryer(g), dry(g, true), \rightarrow dry(g, false)\}$

Notice that the rule generated from (1) and (8) is only generated once. Each rule element has a unique *id* (not shown), and rule element sets are only combined if the last element of the combining rule element set has a higher *id* than the last element of the rule element set it is to be combined with.

aprioriPrune: The *apriorPrune* step deletes candidate rule element sets for which a two rule element subset does not exist. Examining the generated rules:

- The rule generated from (1) & (8) contains the subset *{dry(g,false), dry(g,true)}* which is not present in the level 2 rules. This rule is, therefore, removed by *aprioriPrune*.

In the full data set, this level 2 rule element set is not observed (there is no PDI containing *{dry(g,false),dry(g,true)}* rule elements*)*, because the gripper cannot be dry and wet simultaneously. The *aprioriPrune* function is able to draw this conclusion without a further pass through the data.

The rule generated from rules (3) and (8) is also not present in the data, but all of its subsets are present and it cannot, therefore, be pruned by *aprioriPrune.* Immediately following the *aprioriGen* function call in the ASDD algorithm, the support count of the generated rule element sets is counted via a pass through the PDI data. The rule element set $\{dryer(g), dry(g, true), \rightarrow dry(g, false)\}$ will have support count zero after this data pass and will therefore be removed by step 3 of the ASDD algorithm.

6.4.9 AprioriFilter

The *aprioriFilter* function test for conditional independence is similar to the *filter* function defined in MSDD (see section 3.1.3). It checks candidate rules at level *k* (parameter *Ck*) against rules at level *k-3* (parameter *Lk-3*) to evaluate whether the three additional rule conditions have a significant influence on the probability of the rule's outcome. The *gLevel* parameter defines the G statistic level at which rules are filtered. The G statistic is a statistical test for non-independence (see section 6.4.10).

```
aprioriFilter(Ck, Lk-3, gLevel)
   S = Ck;
   rulesLk-3 = {l ∈ Lk-3 | hasOutcome(l)};
   for all (s ∈ S where hasEffect(s))
      for all (lr ∈ rulesLk-3)
         if (subsumes(lr,s) and gTest(s,lr) < gLevel) {
            remove s from S;
            next s;
         }
   return S;
```

Algorithm 6-6: aprioriFilter. Ck = candidates at level k, Lk-3 = candidates at level k-3, gLevel = g-statistic level for significance tests.

Rules filtered by the *aprioriFilter* function are removed in the same way as pruned rule element sets, and therefore take no further part in rule generation.

If, for example, the rule defined by the level rule element set: $\{b,\rightarrow a\}$ is removed by this method, then no further rules will be generated with head a and body b (e.g. $\{b,c,\rightarrow a\}$ and $\{b,d,\rightarrow a\}$) could not be present in the final rule set (note: the commas indicate that these are sets of rule elements).

The removal of rule element sets in this way can cause a problem when the effect of b as a condition for a is not immediately apparent (e.g. the XOR function in which the output is determined by a combination of each input, with the observation of a single input appearing to have no bearing on the output).

The problem was resolved by setting the significance parameter to 0.445 (50% significance). The standard filter is set at 3.84 for 5% significance, while for low sample sizes 2.706 is used (10% significance). In addition, rules are not filtered using *aprioriFilter* until level 4 (i.e. the rule $\{b,c,d,\rightarrow a\}$ can be filtered by $\{\rightarrow a\}$), and by filtering against rules with three less conditions (*k-3*). The significance levels used in this research for the standard filter process match those used in MSDD [64]. The 50% significance level for the *aprioriFilter* process was chosen to minimise the risk of eliminating useful rule conditions early in the process. Further experimentation in this area is a subject for future work.

The *aprioriFilter* function alters the speed of completion of the rule generation part of the ASDD algorithm when compared to the Apriori algorithm, because rules that are not significant at each level are removed and, as a consequence, generate no children. The Apriori algorithm halts when there are no further rules that can be generated above minimum support. ASDD halts with the additional criteria that there are likely to be no further significant rules.

6.4.10 Conditional Independence

The *filter* and *aprioriFilter* functions use the G statistic [96] to determine conditional independence (defined in appendix section A.2). Intuitively, the method measures the significance of additional rule conditions to the outcome probability of a rule.

The P-SPO set below shows five candidates for the *dryer* P-SPO when acquired empirically from observation data:

$$dryer(g): dry(g,true) \rightarrow \{1.0: dry(g,true)\}$$
$$dryer(g): dry(g,false) \rightarrow \begin{Bmatrix} 0.91: dry(g,true) \\ 0.09: dry(g,false) \end{Bmatrix}$$
$$dryer(g): \{\} \rightarrow \begin{Bmatrix} 0.96: dry(g,true) \\ 0.04: dry(g,false) \end{Bmatrix}$$
$$dryer(g): dry(g,false), painted(b,true) \rightarrow \begin{Bmatrix} 0.92: dry(g,true) \\ 0.08: dry(g,false) \end{Bmatrix}$$
$$dryer(g): dry(g,false), painted(b,false) \rightarrow \begin{Bmatrix} 0.88: dry(g,true) \\ 0.12: dry(g,false) \end{Bmatrix}$$

The first two operators reflect the correct, complete, conditions for the *dryer* action, in that using *dryer* when the gripper is already dry will cause it to remain dry, and using *dryer* when the gripper is wet will cause it to become dry 90% of the time. The 0.91 probability reflects the empirical estimate of the probability from the data (with the actual probability being 0.9).

The third rule has no conditions and gives the probability of finding the gripper dry or wet after a *dryer* action, irrespective of the initial dry state of the gripper. This will be the summed probabilities of all data matching the first two rules.

The fourth and fifth rules contain an additional *painted(b, true)* or *painted(b, false)* condition and the associated probabilities are, again, the matches for the full rule divided by the matches for the rule conditions.

Given the rule set above, the G statistic measure can be used to conclude that the additional *dry(X, true)* and *dry(X, false)* conditions are significant to the outcome probabilities for the *dry(X)* action, but that the *painted(b,true)* or *painted(b, false)* conditions are not significant.

Note that the discussion above is a simplification. The G statistic is a statistical test of non-independence, computed for a 2×2 contingency table of observed occurrences (rather than probabilities).

6.4.11 Extracting One Rule Element Sets from PDIs

PDIs are made up of sets of actions and perceptual features. Each element of the PDI is a ground instance. Single elements are, therefore, simply the extracted features of the data set. The task of the *extractOneRuleElementSets* function is to identify these elements and to assign each a unique identifier to optimise set comparison operations. The function makes a single pass through the database of PDIs, *D*, checking that each element of the PDI is present in the current set of single rule element sets. If it is present, its support count is incremented; otherwise, it is added (with support count 1).

```
extractOneRuleElementSets(D)
   REF = {}; //single rule elements
   L1 = {}; //singe rule element sets
   id = 0;
   for all (pdi ∈ D)
      for all (i ∈ pdi) {
         if (i ∈ L1) {
            R = {r ∈ REF|r.x=i};
            R.sc++; //increment support count of R;
         } else {
            NewR.x = i; //Add a new one rule element set
            NewR.id = id; //one rule element sets have unique id
            NewR.sc = NewR.bs = 1; //bs and sc initialised to 1
            add (L1, NewR);
         }
      }
```

Algorithm 6-7: extractOneRuleElementSets. D=database of perceptual data items.

In addition, the algorithm can identify the possible values of a perceptual feature, by examining the value of extracted single rule elements (axioms) that match in all but the last field (not shown). The syntax definition of perceptual features identifies this as the variable field. For example, a Boolean perceptual feature, such as *dry(g, X)*, where *X* can take values *true* or *false*. This information can, optionally, be given as background knowledge (by the *conflicted* function).

6.4.12 Add Rule Complements

Rule sets required to form a full P-SPO can sometimes be incomplete because:

- The *filter* function can filter rules and not their complements.
- Rules with low probability outcomes can have a support count, *sc,* below *MINSUP*.

For example, for the perceptual feature *painted(g,X)*, *X* can take the values *true* or *false* and the rule generation process can generate the rule element sets:

1) $\{paint(g,b), painted(b,false), holding(b,false), \rightarrow painted(b,true)\}\ pr:0.1$

2) $\{paint(g,b), painted(b,false), holding(b,false), \rightarrow painted(b,false)\}\ pr:0.9$

The filter process could filter rule 1 above, but leave rule 2. This would cause a problem for the successor percept state generation algorithm (section 5.3), because the set of rules will not generate percepts with *painted (b,true)* present.

The *addRuleComplements* function iterates through all rules in learned rule set, *R*, checking that all possible values of each rules outcome are either present in *R* already or do not match any observations in the data *D*. If a missing rule is found, it is added to *R*. The body support, *bs*, of the rule is set to equal the body support of the existing rule because they have the same rule body.

```
addRuleComplements(R, D)
   for (r ∈ R) do
      o = head(r);
      //loop for all possible output values of the rule
      for (oValue ∈ possibleValues(o)) {
         if (oValue ≠ o)
            newRule = copy of r with o replaced by oValue
            if (newRule ∉ R) //if the new rule doesn't exist
               matches = countMatches(newRule, D);
               if (matches ≠ 0) {
                  //it was missed so add it
                  newRule.bs = r.bs; //body support will be the same
                  newRule.sc = matches;
                  R = R ∪ newRule;
               }
      }
```

Algorithm 6-8: addRuleComplements. R = complete rule set. D = database of perceptual data items.

- *possibleValues:* this can be defined by background knowledge, or can be stored in each one-rule element set extracted by the *extractOneRuleElementSets* function (above).

6.5 Create P-SPOs from Rules

The ASDD algorithm creates an individual rule for each outcome set. *P-SPOs* are created by combining these individual rules. The algorithm first searches for rules that have the same conditions, then checks whether they refer to the same perceptual feature in their outcome.

Take, for example, the rule element sets:

1) $\{paint(g,b), dry(g,true), \rightarrow dry(g,true)\}$

2) $\{paint(g,b), dry(g,true), \rightarrow dry(g,false)\}$

These sets have outcome rule elements (indicted by the $\rightarrow$ symbol) and are therefore rules. The rule element sets in rule form are:

$$paint(g,b):dry(g,true)\rightarrow\{0.4:dry(g,true)\}$$
$$paint(g,b):dry(g,true)\rightarrow\{0.6:dry(g,false)\}$$

These rules are combined by the *createP-SPOs* algorithm to form the P-SPO:

$$paint(g,b):dry(g,true)\rightarrow\left\{\begin{matrix}0.4:dry(g,true)\\0.6:dry(g,false)\end{matrix}\right\}$$

P-SPOs are sets of rules with the same conditions (rule body) which apply to the same perceptual feature in their outcome.

createPSPOs takes as input a set of rules, *R*, and possible actions, A, and returns a set of P-SPOs, *P*. The algorithm iterates through all rules, checking each one against all other rules for which: the body of the rule matches, and the head of the rule (outcome element) refers to the same perceptual feature. If the rules pass this test then they are part of the same P-SPO set. The P-SPOs are then created by combining the rules contained in the P-SPO sets.

```
createPSPOs(R, A)
   PS = {}; //list of rules in same PSO set
   P = {}; list of PSOs

   //cycle through all rules adding them to PSPO sets
   //if they have matching bodies and conflicting outcomes
   for (r ∈ R) {
      if (r.pSet ≠ {})        //r already part of a PSPO
         next r;
      for all (c = next(r); c ∈ R) {
         if (body(r) ≠ body(c))
            next c;           //not the same body
         if (head(r) ∉ possibleValues(head(c)))
            next c;           //no conflicting outcome
         //these rules are parts of the same PSPSO
         r.pSet = r.pSet ∪ c; //add c to the PSPO set
         c.pSet = r.pSet;     //c refers to PSPO of r
         if (r.pSet ∉ PS)     //PSPO not previously defined
            PS = PS ∪ r.pSet; //add it to the set of PSPOs
      } //next c
   } //next r

   //construct the PSPOs from the rule sets
   for (ps∈PS) {
      P.add(newPSPO);
      newPSPO.P_A={first(ps).x|x∈ A}; //set the action
      //context is rule elemenents without head and action
      newPSPO.P_C=first(ps).x minus {newPSPO.P_A ∪head(ps)};
      for all (pi∈ps) {
         add(newPSPO.P_O,head(pi));   //add each outcome
         add(newPSPO.P_P,pi.sc/pi.bs);//and associated probability
      }
   }
   return P;
```

Algorithm 6-9: createPSPOs. R = complete rule set. A = set of possible agent actions. The algorithm returns P, a set of P-SPOs build from the rule set.

The final section of the algorithm creates the P-SPOs, such that each contains:

- *An action,* P_A: the action contained in the rule element set. All rules in the set have the same context, so the first rule is used to determine this.
- *A context,* P_C: the rule element set with the action and the outcome removed. Again, all rules in the set have the same context, so the first rule is used.
- *A set of outcomes,* P_O: the heads of each the rules, added in order.
- *A set of probabilities,* P_P: the support count for the rule divided by the body support for the rule, added in the same order as P_o.

6.6 Establishing P-SPO Precedence

The *precedence* algorithm provides a method for resolving conflicts when P-SPOs matching the input percept have conflicting outcomes. A conflicting outcome occurs when the outcome refers to the same perceptual feature (as discussed in section 5.3.3). The algorithm establishes

which P-SPOs defer to, and which P-SPOs have precedence over, other P-SPOs in the case of a conflict.

Conflicting operators can be the result of a partial model being acquired from the data by ASDD due to a small learning data set, or can be the result of operator conditions which give differing evidence of the outcome.

Two P-SPOs are shown below:

$$new(b) : painted(b, false) \rightarrow \{1.0 : painted(b, false)\}$$
$$\{\} : \begin{matrix} painted(b, false), \\ holding(b, false) \end{matrix} \rightarrow \begin{Bmatrix} 0.02 : painted(b, true) \\ 0.98 : painted(b, false) \end{Bmatrix}$$

The first of the above operators is the P-SPO for the *new* action's effect on the *painted(b,X)* perceptual feature. The second P-SPO is a partial rule, which could be created if there is not enough evidence to correctly learn the rule for the *paint* action from the original rule-set:

$$paint(X, Y) : \begin{matrix} painted(Y, false), \\ holding(Y, false) \end{matrix} \rightarrow \begin{Bmatrix} 0.1 : painted(Y, true) \\ 0.9 : painted(Y, false) \end{Bmatrix}$$

Conflict resolution for these operators would need to occur if, for example, the action *new(b)*, was chosen for an initial percept:

$$painted(b, false),\ dry(g, true),\ holding(b, false), reward(none)$$

The *precedence* algorithm defines how conflicts of this type are resolved. The algorithm evaluates the precedence of a generated set of P-SPOs, *PSPOs*, over a set of PDIs, *D*.

All PDIs in the database are examined. If two P-SPOs apply to the same PDI and refer to the same perceptual feature in the outcome, the operator precedence is defined using the *firstPSPOSuperior* function, which finds the subset of PDIs for which both rule sets apply and uses a heuristic error measure to define the operator with the most accurate performance for the subset (section 6.6.1).

```
precedence(PSPOs, D)
   for (pdi ∈ D){
      matchedPSPOs = matching(PSPOs, pdi);
      for all (pspo1 ∈ matchedPSPOs) {
         for (pspo2 = next(pso1); pso2 ∈ matchedPSPOs) {
            if (!conflicted(pspo1.P_O ∪pspo2.P_O))
               next pso2;
            if (precedenceSet(pso1, pso2)
               next pso2;
            if (firstPSPOSetSuperior(pso1, pso2))
               setPrecedenceOver(pos2, pso1);
            else
               setPrecedenceOver(pso1, pso2);
         }
      }
   }
```

Algorithm 6-10: precedence. PSPOs=the operator set. D = perceptual data items. The algorithm sets the precedence between all operators. Precedence defines which operator will be used if there is a conflict.

1) *D* can be either the same set of data used to learn the operators, or a separate set used to establish precedence between operators. If the same data set is used, the speed of the algorithm can be increased by the observation that a specific rule set (one which matches fewer examples in the PDI set) will always have precedence over a general one according to the error measure used (section 6.6.1).

2) The *matching* function returns the subset of P-SPOs with a body matching the percept and action from the PDI. This is similar to the subset function defined for rule element sets in section 6.4.7, but is defined for P-SPOs (which have multiple outcomes).

```
matching(PSPOs, PDI)
   subsetP = {};
   for all (p ∈ PSPOs)
      for all (po ∈ p.P_O)
         if ((c.P_O ∈ PDI) and (c.P_C ⊂ PDI)){
            subsetP.add(p);
            next p;
         }
   return subsetP;
```

Algorithm 6-11: matching. PSPO = the planning operator set. PDI = a single perceptual data item. The algorithm returns the subset of planning operators with context and action matching the PDI.

3) The *conflicted* function is as defined by background knowledge and returns *true* if the outcome sets of the two P-SPOs have conflicting elements (see section 5.2.1).

6.6.1 First PSPO Superior

The *firstPSPOSuperior* function returns true if the first P-SPO should have precedence in situations where the two P-SPOs are in conflict (apply to the same outcome perceptual feature).

```
firstPSPOSuperior(PSPO1, PSPO2, D)
   //check if the rule is already part of the defers set
   if (PSPO2 ∉ PSPO1.P_D)
     return true;
   //create a new combined PSPO
   newC.P_A =PSPO1.P_A;          //action must be the same
   newC.P_C = PSPO1.P_C ∪PSPO2.P_C; //conditions are combined
   newC.P_O = PSPO1.P_O ∪PSPO2.P_O; //outcomes are combined
   newC.P_P = PSPO1.P_P ∪PSPO2.P_P; //set size of outcome set

   //find body support for PSPO and support count for outcomes
   sc[|newC.P_O|] = bs = 0;
   for all (pdi ∈ D)
     if (newC.P_C ∪newC.P_A ∈ pdi){
       bs++; i = 0;
       for all (o ∉ newC.P_O) {
          if (o ∈ pdi)
            sc[i]++;
          i++;
       }
     }
   //set the probabilities of the combined outcome set
   for (i = 0 to |newC.P_O|)
      newC.P_P[i] = sc[i]/bs;
   //find rule with the least error against the combined set
   if (errorMeasure(PSPO1, newC)≤errorMeasure(PSPO2, newC)){
      PSPO2.P_D = PSPO2.P_D ∪PSPO1;
      return true;
   }
   return false;
```

Algorithm 6-12: firstPSPOSuperior. PSPO1 and PSPO2 = the planning operators to be tested. D = the set of perceptual data items. The algorithm return true if the first P-SPO will have precedence in situations where the rules are in conflict.

The algorithm compares the probability values for the outcomes of the P-SPOs with a new P-SPO generated by combining the conditions of the operators. The probabilities for the new P-SPO (which are collections of rule sets) are generated empirically in the same manner as all other rule sets (section 6.4.10) *prob = sc/bs.*

The P-SPO that has the least error when compared to the combined P-SPO is given precedence. The error measure used in this research was introduced by the author in [18]:

- *For each non-matching outcome:* add +0.5. A non matching outcome is one that is present in the combined rule but not in the original rule, or present in the original rule but has probability zero in the combined rule.
- *For each matching outcome:* add the absolute difference between the empirical probability for the combined rule and that for the original rule.

The use of the combined outcomes provides a measure of the accuracy of each operator in situations for which the P-SPOs conflict. This addition of 0.5 for missing/additional outcomes to the error measure penalises rules which failed to generate all outcomes for a P-SPO, however low the probability of the outcome.

Note 1: For a rule set which is *subsumed* by a more general rule set, the specific rule set will always have precedence over a general one, if we are using the same data set to test rule sets as to create them. This is because the combined rule set will be equal to the more specific rule set. For example, if we have a rule with conditions {a,b} and a rule with conditions {a}, the combined rule has conditions {a,b}.

Note 2: If the combined rule set applies to a limited number of examples from the data this method is likely to produce spurious results.

Example 1:

The P-SPOs (1) and (2) below are generated from data and have the condition (non-outcome) rule elements *{paint(g,b), painted(b,false)}* and *{paint(g,b), holding(b,false)}* respectively:

1) $$paint(g,b) : painted(b, false) \rightarrow \begin{Bmatrix} 0.31 : painted(b, true) \\ 0.79 : painted(b, false) \end{Bmatrix}$$

2) $$paint(g,b) : holding(b, false) \rightarrow \begin{Bmatrix} 0.42 : painted(b, true) \\ 0.58 : painted(b, false) \end{Bmatrix}$$

Combining the two sets of conditions and the associated outcomes (rule heads) gives the new P-SPO (initially unknown probabilities indicated by question marks):

$$paint(g,b): \begin{matrix} painted(b,false), \\ holding(b,false) \end{matrix} \rightarrow \left\{ \begin{matrix} ?: painted(b,true) \\ ?: painted(b,false) \end{matrix} \right\}$$

If the conditions of this combined P-SPO are equal to the conditions of one of the P-SPOs in the P-SPO set generated by ASDD, then the probabilities associated with each outcome can be taken from the operator, otherwise, a new pass through the data will be required to find the empirical probabilities of the combined operator.

In this instance, an operator with the combined conditions is likely to have been discovered by ASDD, given a reasonably large data set, and the associated outcomes and probabilities can be found from that operator:

$$paint(g,b): \begin{matrix} painted(b,false), \\ holding(b,false) \end{matrix} \rightarrow \left\{ \begin{matrix} 0.09: painted(b,true) \\ 0.91: painted(b,false) \end{matrix} \right\}$$

Using the error measure, rule (1) has an error of:

$|0.31-0.09| = 0.22$: for *painted(b,true)* +

$|0.79-0.91| = 0.12$: for *painted(b, false)*

Total error for rule (1): 0.34.

Using this error measure, rule (2) has an error of:

$|0.42-0.09| = 0.33$: for *painted(b,true)* +

$|0.58-0.91| = 0.33$: for *painted(b, false)*

Total error for rule (2): 0.66.

Rule (1) would, therefore, have precedence over rule (2).

Example 2

A similar example illustrating the error measure for non-matching outcomes is given below for the paint rule when the robot is holding the block.

The P-SPOs (1) and (2) below are generated from data and have the condition (non-outcome) rule elements *{paint(g,b), painted(b,false)}* and *{paint(g,b), holding(b,true)}* respectively:

1) $paint(g,b): painted(b,false) \rightarrow \left\{ \begin{matrix} 0.31: painted(b,true) \\ 0.79: painted(b,false) \end{matrix} \right\}$

2) $paint(g,b) : holding(b,true) \rightarrow \{1.0 : painted(b,true)\}$

Combining the two sets of conditions and the associated outcomes gives the new P-SPO:

$$paint(g,b) : \begin{matrix} painted(b,false), \\ holding(b,true) \end{matrix} \rightarrow \begin{Bmatrix} ? : painted(b,true) \\ ? : painted(b,false) \end{Bmatrix}$$

In this instance, ASDD will not have produced a P-SPO with the combined conditions, because the rule element sets needed to form the operator would have been filtered due to the extra *painted(b,false)* condition having no significant effect on the probability of rule (2) above.

The probabilities of the combined operator can be found from data (or by adjusting the algorithm to take probability 1.0 subsets into account). The combined operator probabilities will be found to be:

$$paint(g,b) : \begin{matrix} painted(b,false), \\ holding(b,true) \end{matrix} \rightarrow \begin{Bmatrix} 1.0 : painted(b,true) \\ 0.0 : painted(b,false) \end{Bmatrix}$$

Using the error measure, rule (1) has an error of:

$|0.31 - 1.0| = 0.79$: for *painted(b,true)* +

0.5 : for the *painted(b, false)* outcome which was not present in the combined PSPO

Total error for rule (1): 1.29.

Rule (2) has an error of:

$|1.0 - 0.0| = 0.0$: for *painted(b,true)* +

0 : for *painted(b, false)* which was (correctly) not present in the combined PSPO

Total error for rule (2): 0.0

Rule (2) would, therefore have precedence over rule (1).

Special Situations:

There are some exceptions to choosing the rule with the lowest error measure, which help to keep the model accurate in the general case, and help the RVRL algorithm (chapter 9) to gather useful information.

The following special conditions are applied in order:

1) If the P-SPOs have an equal error measure (e.g. they both have a single outcome with probability 1.0), the *more general* rule is given precedence because it has more supporting evidence.

2) If the P-SPOs being compared are both deterministic (probability 1.0) and one is a "frame rule", then the frame rule is given supremacy, because rules of this type can be combined into environment operators.

3) If each of the outcomes contained in the two P-SPOs are conditionally independent (using the G statistic measure), the *more general* one is used because, again, the general rule has more supporting evidence.

A simple definition of *more general* is: the P-SPO with the most conditions. However, some P-SPOs may contain rare conditions which make them more specific. Generality can, instead, be induced from the training data, by using body support, *bs*: the most general P-SPO is the one whose conditions match the greatest number of PDIs from the training data.

Note: To aid comprehension, these exceptions are not shown in the *firstPSPOSuperior* algorithm (algorithm 6-12).

6.6.2 Outcome Sets of Size greater than one

Outcome sets with size one (dependent outcomes) will always take precedence over single outcome sets, because the single outcome operators that they are generated from must have equal conditions. Generation of rules of this type is beyond the scope of this work, but is discussed in the future work (section 11.3.5). There are two methods: (i) the rules can be generated by ASDD, which would require the parent rules to be matching in all but one element (the additional outcome); (ii) an additional pass can be made through the rules for those that have the same conditions, and have outcomes for which the additional dependency of the outcomes passes the G-test.

This equality in conditions means that that both parent rules can be safely set to defer to the combined rule because they cover the same situations.

6.7 ASDDs: Speeding up ASDD with Set Operators

The ASDDs algorithm is an optimisation of ASDD which increases the speed of the support count operation by using set operations to find intersections between the PDIs covered by each rule-element set. There is an overhead in terms of the storage required by each rule because each single item set must be associated with the PDIs that it is present in. The optimisation is an adaptation of the *AprioriTID* algorithm [1], using a similar method of set

counting, but associating each single item set with the PDIs it contains to perform set counts. Extensions to overcome overheads in memory requirements are beyond the scope of this work, but could be achieved by partitioning the PDI database into smaller sets (e.g. by use of the *Partition* algorithm [81]).

Set intersections can be used to count item-sets in the following way (explanation adapted from [42]).

A PDI reference is a unique identifier for a perceptual data item. For a *single rule element set* (set of size one), the PDI list is the set of identifiers corresponding to the PDIs in which it is present. Each rule element set (sets of size > 1) also has a corresponding PDI list. The PDI list for a newly generated candidate *C*, generated from parent rule element sets *X* and *Y*, has a PDI list equal to the intersection of the PDI list for *X* and the PDI list for *Y*. The support count of the candidate is equal to number of items in the PDI list.

Storage required for a PDI list for each rule element set quickly becomes an issue as the database becomes large and the candidate set increases. The PDI lists can be generated on the fly by keeping track of the parents of each rule element set. Each parent's PDI list can then be generated as an intersection of the PDI's of its parents until a stored PDI list it found (e.g. the PDI set for *a single rule element* set). The method used in this research was to store PDI list for each *two rule element set*. This offered the best compromise between storage and speed. This was not a focus of the research and further optimisations are a subject for future work.

Steps of the algorithm:

1) On the initial pass of the database, used to discover size-one rule element sets in the standard algorithm, each level one rule element set stores a set of PDIs that it appears in. The size of this set is the support count.

2) The *aprioriPrune* algorithm is amended such that a reference to the new rule's parents is recorded by the surviving (non-pruned) candidates. If this is a two item set, the generated candidate's PDI set is equal to the intersection of the parent PDI set and the support count is the size of the PDI set. If the rule set is of size > 2, the PDI set is generated as the intersection of the parent's PDI sets. Support count is the size of the intersection. The new PDI set is not stored for size > 2 sets.

```
aprioriPrune(Ck, Lk-1)
   for all (c ∈ Ck)
      forall (k-1 size subsets s of c)
         if (s ∉ Lk-1)
            delete c from Ck;

   for all (c ∈ Ck)
      PDIset = c.mother.PDIset ∩ c.father.PDIset
      c.sc = |PDIset|;
      if (|c.x| == 2)
         c.PDIset = PDIset;
```

Algorithm 6-13: aprioriPrune. Modified for the ASDDs optimisation for ASDD.

Support count has been counted in advance (the size of the PDI set). Step 2 of the ASDD algorithm is, therefore, no longer necessary and is removed. The algorithm is otherwise unchanged.

6.8 Summary

This chapter defined the ASDD algorithm for the fast generation of stochastic rules from a database of perceptual data, and the support algorithms required for generation of P-SPOs from these rules. Precedence between operators is established for situations in which the rules conflict. Finally an optimisation to the algorithm using set-based techniques was given.

7. Test Environments

This chapter defines the test environments for both the ASDD algorithm (chapter 6) and RVRL algorithm (chapter 9) including characteristics and challenges that these test environments present. These test environments are used to evaluate the performance of ASDD in chapter 8 and the performance of RVRL in chapter 10.

Two test environments were selected to evaluate the performance of the system under a range of conditions. The "slippery gripper" environment is fully observable and all outcomes are independent. The environment can be completely defined by a set of PSPOs. The "predator-prey" environment is partially observable, contains independent outcomes, and changes outside the direct control of the agent. An accurate P-SPO set for the environment cannot be learned by ASDD, but an approximation can be acquired.

7.1 The Slippery Gripper Environment

The "slippery gripper" test environment presented is an adaptation of the environment defined by Oates and Cohen [65]. The environment has an additional "gripper clean" environment feature from the one used in the explanation of P-SPO operators given in section 5.2 and has slightly increased complexity in its dynamics.

This is a discrete time step environment. The environment changes state in response to each selected action with no external events. The environment is, therefore, completely defined by the actions available to the agent.

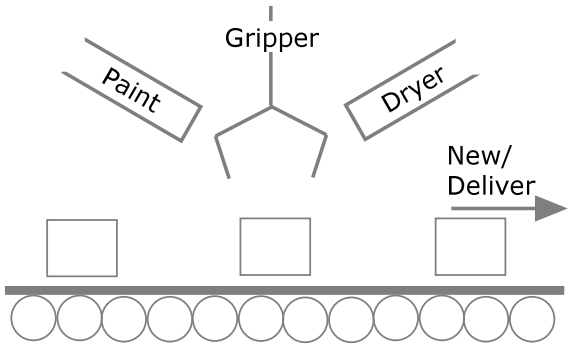


Figure 7.1: The "slippery gripper" environment.

- ***Paint:*** paints blocks.
    - If the robot is holding the block then the block will become painted and the gripper will become dirty 100% of the time.

- If the robot is not holding the block then the block will become painted 10% of the time and the gripper will become dirty 20% of the time.
- If the gripper was clean and the block was not held then the gripper becomes dirty 20% of the time.
- If the gripper was clean and the block was held then the gripper becomes dirty 100% of the time.

- ***Dryer:*** dries the gripper.
  - If the gripper was wet then it will become dry 90% of the time.
- ***Pickup:*** picks up blocks.
  - If the gripper is dry, the block was not held, and the block was not painted then the action results in the block being held 95% of the time.
  - If the gripper is dry, the block was not held, and the block was painted then the action results in the block being held 75% of the time.
  - If the gripper is not dry, the block was not held and not painted then the block will be held 15% of the time.
  - If the gripper is not dry, the block was not held and not painted then the block will be held 5% of the time.
  - If the gripper was clean and the block was painted then it becomes dirty 20% of the time.
- ***New****:* used to deliver a block and receive a new one.
  - The action results in a positive reward if the block was painted and a negative one if it was not.
  - A new block arrives which is clean, and not painted or held in the gripper.
  - The gripper will be dry after the action is completed 30% of the time and is wet the remaining 70% of the time.

The sensors are defined as:

$$S = \{BP, GC, GD, HB, R\}$$
$$S_{BP} = \{BP, \neg BP\}$$
$$S_{GC} = \{GC, \neg GC\}$$
$$S_{GD} = \{GD, \neg GD\}$$
$$S_{HB} = \{HB, \neg HB\}$$
$$S_{R} = \{pos, neg, none\}$$

Where:

BP = block painted;

GC = gripper clean;

GD = gripper dry;

HB = holding block;

R = reward.

The *sense* function returns the current state of each element of the sensor array.

The agent's *perceive* function converts the sensor information into a percept. In this case this is a simple one-to-one mapping between the sensor information and the percept. The background knowledge required to describe this environment is given below:

$$reward(X) \leftarrow X \in \{pos, neg, none\}$$
$$boolean(X) \leftarrow X \in \{true, false\}$$
$$painted(X) \leftarrow boolean(X)$$
$$dry(X) \leftarrow boolean(X)$$
$$holding(X) \leftarrow boolean(X)$$
$$clean(X) \leftarrow boolean(Y)$$
$$conflicted(P) \leftarrow painted(true) \in P, painted(false) \in P$$
$$conflicted(P) \leftarrow dry(true) \in P, dry(false) \in P$$
$$conflicted(P) \leftarrow holding(true) \in P, holding(false) \in P$$
$$conflicted(P) \leftarrow clean(true) \in P, clean(false) \in P$$
$$conflicted(P) \leftarrow reward(X) \in P, reward(Z) \in P, X \neq Z$$

Figure 7.2: Background knowledge for the "slippery gripper" environment with additional "clean" perceptual feature.

The background knowledge is similar to that given in section 5.2 with the addition of the *clean* perceptual feature.

Figure 7.1 presets a situation in which the robot is not holding the block. If the block has not been painted (S_{BP}=¬BP), the gripper is wet (S_{GD}=¬GD), the gripper is clean (S_{GC}=GC) and no reward was received (S_R = none) in the previous time step, the sensor information can be converted into a percept given by:

painted(false), dry(false), clean(true), holding(false), reward(none)

The key features of the test environment are:

- It is a fully observable Markov decision process (all features of the environment can be observed, and the next state is dependent only on the previous state).
- The perceptual features of the environment are conditionally independent (the values of the features in the successor state are not dependent on the values of other features in the successor state).
- The environment is stationary (the model does not change over time).
- The environment is continuous (the task is assumed to continue infinitely, as opposed to episodic tasks, which are re-started after an absorbing state has been reached).

These features mean that it is possible to model the environment perfectly with a set of P-SPOs, and it should, therefore provide a useful benchmark for performance tests of the algorithm.

The P-SPO set for the standard environment is amended from those given in section 5.2. The P-SPO set does not require variables for the gripper and block, because it assumed that there is only ever one current block and one gripper (matching the environment definition above). There is also an additional *clean* environment feature and altered probabilities for the outputs. Figure 7.3: The P-SPO set for a “slippery gripper” environment with exactly one block and one gripper.

$$paint: holding(true) \rightarrow \{1.0: painted(true)\} \quad (1)$$

$$paint: holding(true) \rightarrow \{1.0: clean(false)\} \quad (2)$$

$$paint: holding(false) \rightarrow \begin{Bmatrix} 0.8: clean(true) \\ 0.2: clean(false) \end{Bmatrix} \quad (3)$$

$$paint: \begin{matrix} painted(false), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.1: painted(true) \\ 0.9: painted(false) \end{Bmatrix} \quad (4)$$

$$paint: \begin{matrix} clean(true), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.2: clean(true) \\ 0.8: clean(false) \end{Bmatrix} \quad (5)$$

$$paint: holding(true) \rightarrow \{1.0: clean(false)\} \quad (6)$$

$$dryer: dry(false) \rightarrow \begin{Bmatrix} 0.9: dry(true) \\ 0.1: dry(false) \end{Bmatrix} \quad (7)$$

$$pickup: \begin{matrix} dry(true), painted(false), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.95: holding(true) \\ 0.05: holding(false) \end{Bmatrix} \quad (8)$$

$$pickup: \begin{matrix} dry(true), painted(true), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.75: holding(true) \\ 0.25: holding(false) \end{Bmatrix} \quad (9)$$

$$pickup: \begin{matrix} dry(false), painted(false), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.15: holding(true) \\ 0.85: holding(false) \end{Bmatrix} \quad (10)$$

$$pickup: \begin{matrix} dry(false), painted(true), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.05: holding(true) \\ 0.95: holding(false) \end{Bmatrix} \quad (11)$$

$$pickup: \begin{matrix} painted(true), \\ clean(true) \end{matrix} \rightarrow \begin{Bmatrix} 0.20: clean(true) \\ 0.80: clean(false) \end{Bmatrix} \quad (12)$$

$$new: painted(false) \rightarrow \{1.0: reward(neg)\} \quad (13)$$

$$new: painted(true) \rightarrow \{1.0: reward(pos)\} \quad (14)$$

$$new: \{\} \rightarrow \{1.0: holding(false)\} \quad (15)$$

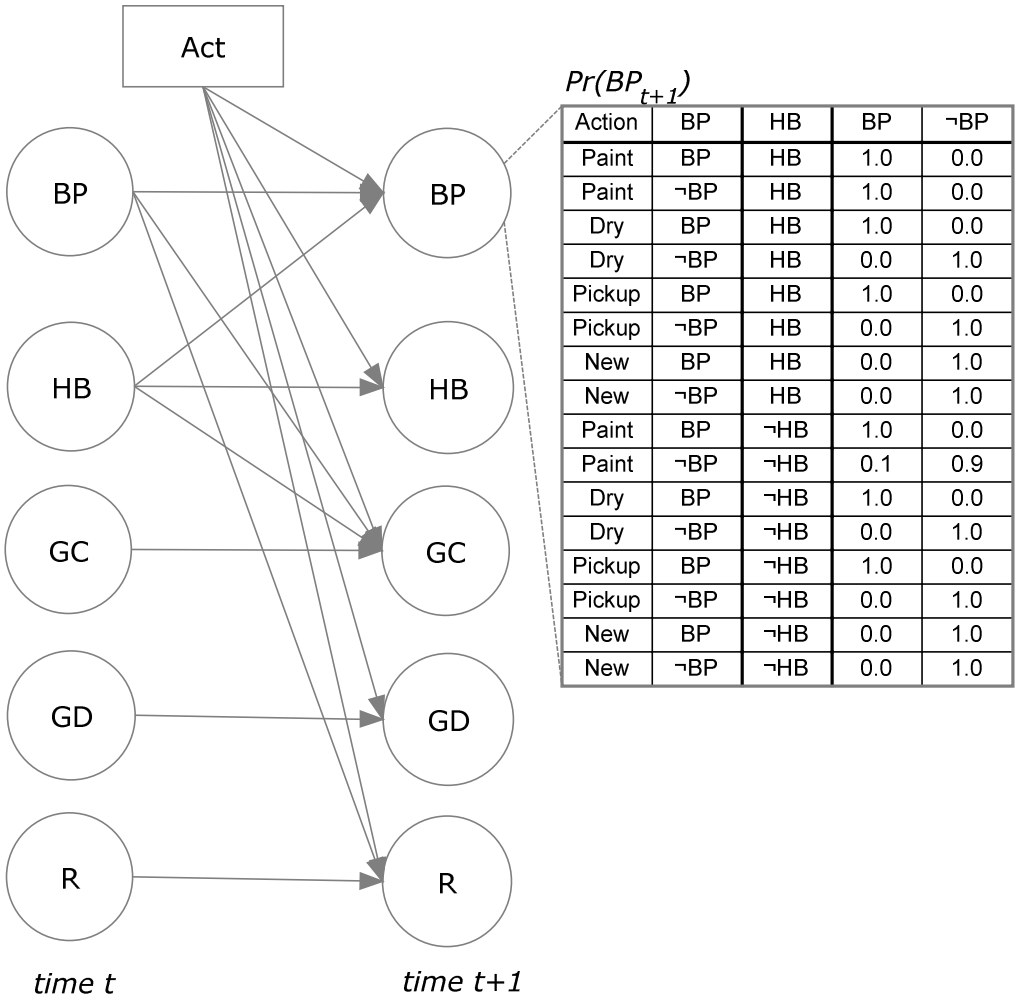
$$new: \{\} \rightarrow \{1.0: painted(false)\} \quad (16)$$

$$new: \{\} \rightarrow \begin{Bmatrix} 0.3: dry(true) \\ 0.7: dry(false) \end{Bmatrix} \quad (17)$$

$$environment: \{\} \rightarrow \{1.0: reward(none)\} \quad (18)$$

Figure 7.3: The P-SPO set for a "slippery gripper" environment with exactly one block and one gripper.

Figure 7.4 shows the influence diagram for the environment. The diagram highlights dependencies between variables in the domain.



$Pr(BP_{t+1})$

Action	BP	HB	BP	¬BP
Paint	BP	HB	1.0	0.0
Paint	¬BP	HB	1.0	0.0
Dry	BP	HB	1.0	0.0
Dry	¬BP	HB	0.0	1.0
Pickup	BP	HB	1.0	0.0
Pickup	¬BP	HB	0.0	1.0
New	BP	HB	0.0	1.0
New	¬BP	HB	0.0	1.0
Paint	BP	¬HB	1.0	0.0
Paint	¬BP	¬HB	0.1	0.9
Dry	BP	¬HB	1.0	0.0
Dry	¬BP	¬HB	0.0	1.0
Pickup	BP	¬HB	1.0	0.0
Pickup	¬BP	¬HB	0.0	1.0
New	BP	¬HB	0.0	1.0
New	¬BP	¬HB	0.0	1.0

Figure 7.4: Influence diagram showing dependencies between variables for the "slippery gripper" environment. The conditional probability (CPT) table for the BP variable has been included. Other CPTs (omitted for brevity) would follow a similar format.

7.1.1 Notes on the Slippery Gripper Environment

The environment is challenging for a reinforcement learning algorithm because slight differences in the estimations of state action values can result in sub-optimal performance. For example:

- A strategy of painting a block without picking it up, then delivering the block as soon as it has been painted leads to the quickest path to an immediate reward, but with a low probability of success and therefore will not achieve the maximum reward in the long run.

- A strategy of picking up a block, painting it and then delivering it will lead to a marginally better level of reward, but the pickup action often fails if the gripper is wet. In addition, the value of a strategy may be over-estimated or underestimated if the available evidence of state transition probabilities is limited.

- The optimal strategy is to dry the gripper if it is wet. Pickup the block when the gripper is dry, then paint the block and deliver it. Despite having the most steps, this strategy has the highest probability of delivering a painted block per action used.

- The *clean* perceptual feature has no influence on the reward function, but the rule learning mechanism will attempt to learn it's dynamics because the system does not relate rule-learning to rewards.

7.2 The Predator Prey Environment

The predatory-prey environment consists of a 4x4 grid surrounded by a "wall". There is *one* predator agent and *one* prey agent which take simultaneous moves. The predator catches the prey (gains a reward) if the predator and prey are on the same square. The prey selects a random action at each move.

Both predator and prey have four available actions: *move(north), move(east), move(south)* and *move(west)*. An action has the effect of moving the agent one square in the selected direction, unless there is a wall, in which instance the action has no effect.

The environment is continuous: the predator and prey continue to move after the predator catches the prey.

The agent body's *sense* function detects the contents of the four squares adjoining it and the square under it. Each square can be either empty, contain an agent, or contain a wall. Squares can be in only one of these states (the agent does not see its own body in this instance). Figure 7.5 gives an example situation in which the predator has a wall to the west and a prey to the east.

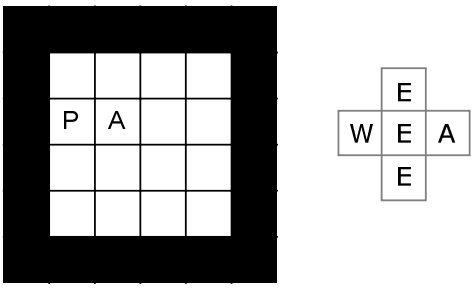


Figure 7.5: Predator and prey in a 4×4 grid (P = predator agent; A= prey agent). The sensor information for the predator, P, is shown to the right (W = wall. E = empty, A = prey agent).

The sensors are defined as:

$S = \{N, E, S, W, U\}$

$S_N = \{Empty\_N, Wall\_N, Agent\_N\}$

S_E, S_S, S_W, S_U follow the same form as S_N

Where:

S_N = see north, and can take the values:

Empty_N: the square to the north is empty.

Wall_N: the square to the north contains a wall.

Agent_N: the square to the north contains an agent.

S_S, S_W and S_U are similarly defined.

The agent's *perceive* function converts the sensor information into a percept. In this case this is a simple one-to-one mapping between the sensor information and the percept. For example a value for S_N of Empty_N converts to the perceptual feature *see(north, empty)*. In accordance with the P-SPO definition, the last parameter of a perceptual feature defines the value, meaning *see(north, X)*, can only take one value for X in a given percept.

The background knowledge required to describe this environment is given below:

$$\begin{aligned} &item(X) \leftarrow X \in \{wall, empty, agent\} \\ &direction(X) \leftarrow X \in \{north, east, south, west, under\} \\ &see(X,Y) \leftarrow direction(X), item(Y) \\ &conflicted(P) \leftarrow see(X,Y) \in P, see(X,Z) \in P, Y \neq Z \end{aligned} \tag{7.1}$$

The background knowledge states that the agent can see the square contents in each of the available directions, and that each square can contain exactly one item.

Figure 7.5 contains an example of sensor information, which can be converted into a percept given by:

$$see(north, empty),\ see(east, agent), see(south, empty), see(west, wall), see(under, empty)$$

The key features of the test environment are:

- It is partially observable (not all features of the environment are contained in the percept)
- The probability of a successor percept is not completely dependent on the previous percept and action selected (the environment does not exhibit the Markov property). An increased history would improve the probability estimates of the following states.
- The perceptual features of the environment are conditionally dependent (the values of the features in the successor state are dependent on the values of other features in the successor state). This requires the additional definition of an *invalid(P)* function for the environment (as defined in section 5.4.5).
- The environment is stationary (the model does not change over time).

The predator-prey environment does not display the Markov property and it is therefore not possible to represent it accurately as an influence diagram or as a set of P-SPOs. The P-SPO set learned by ASDD will be an approximation of the environment's dynamics.

The *invalid(P)* function for the predator prey environment (7.2) eliminates successor percepts in which more than one agent is present, or for which walls are present in opposite directions.

$$\begin{aligned} &invalid(P) \leftarrow see(X, agent) \in P, see(Y, agent) \in P, X \neq Y \\ &invalid(P) \leftarrow see(north, wall) \in P, see(south, wall) \in P \\ &invalid(P) \leftarrow see(east, wall) \in P, see(west\ wall) \in P \end{aligned} \tag{7.2}$$

7.2.1 Notes on the Predator-Prey Environment

The predator prey environment is challenging to the ASDD algorithm because it has a range of features which are outside the modelling capabilities of the basic algorithm. In order to form a perfect model, the algorithm requires that the Markov property holds, that the environment is fully observable and that output variables are independent. Using the algorithm in an environment for which these requirements do not hold shows how the algorithm can perform (to some extent) in these circumstances by using probabilities to model external factors.

Observation of an optimal policy for this environment reveals that the best strategy is to keep the prey agent visible at all times:

- If on top of the prey and next to a wall – move into the wall.
- If the prey agent is in the percept, but not underneath the prey, move onto the prey so that it will remain in sight in the next move
- If on top of the prey and not next to a wall – all moves are equal
- If the prey agent is not in sight and the predator is next to a wall – move into space so that more squares are visible

This is a particularly challenging environment for the ASDD learning algorithm in its current implementation, because the assumption of independent outcomes means that it is not able to predict one of the key features of the environment: that moving onto a prey will always result in the prey being somewhere within the predator's percept.

7.3 Summary

This chapter presented the test environments used to evaluate the partial agent learning framework using a P-SPO set acquired by ASDD (chapter 8) and the full agent learning framework with the addition of the RVRL algorithm (chapter 10).

The "slippery gripper" environment is fully observable and can be completely defined by a set of action rules because all outcomes are (stochastically) defined by the actions of the agent. Additionally, each outcome is independent. It should, therefore, be possible to learn a completely accurate P-SPO definition of the environment using ASDD given a large enough data set.

The "predator-prey" environment is more challenging because it contains independent outcomes, is partially observable, and the environment changes outside the direct control of the agent. These features have the effect that an accurate P-SPO set cannot be learned by ASDD. An approximation can, however, be acquired.

RVRL will attach utility estimate values to the P-SPOs generated for these environments, and the challenges presented by each environment in this respect will be discussed in sections 9 and 10.

8. Performance Results: Agent Framework with ASDD

This chapter presents the results of the ASDD and ASDDs (ASDD with set optimisation) algorithms in learning P-SPO sets for the test environments presented in chapter 7. The tests examine the validity of the learned model in comparison to both a tabular method and rule sets learned by MSDD. The tests also examine the performance of the model when used to derive a policy for the test environment in the batch processed environment modelling and learning framework described in chapter 4.

In this context, performance measures test:

- The ASDD algorithm's capacity to accurately learn a set of parallel stochastic planning operators which model the test environment.
    - Model accuracy is defined in terms of the model's ability to predict future percept probabilities given an initial percept and action when compared to a perfect model.
    - The accuracy of the model provided by the operator set learned by ASDD is compared to the model learned by tabular methods and by the MSDD algorithm, given varying levels of environmental experience.
    - The "slippery gripper" environment is defined in terms of a P-SPO set. The learned P-SPOs can, therefore, be compared to the given operators. For the predator-prey environment, an exemplar of the acquired operators is discussed.
- The speed of the ASDD and ASDDs algorithms in learning operator sets is compared to MSDD for each of the test environments with various levels of environmental experience.
- The performance of the policy derived from an ASDD based model in the test environment, given a limited training data sample, is compared to that of a policy derived from a tabular model, and that provided by an MSDD operator set. All models are derived, and policies formed, using the *BatchModelQ* framework described in chapter 4. Tests are performed to evaluate:
    - The agent's ability to achieve goal states and avoid *disaster* states. A disaster state is one which provides poor or possibly catastrophic performance, such as delivery of an unpainted block in the "slippery gripper" environment.

- The agent's ability to maximise the rewards it can gather from the environment over a test period.

When learning the model, ASDD minimum support was set to 1 (any occurrence means a rule set is not discarded), and significance in *aprioriFilter* was set to 0.445 (50% significance). For both ASDD and MSDD, the G level for the *filter* function was set to 3.841 (5% significance).

8.1 Performance comparisons

The following sections define the learning methods used in performance comparisons.

8.1.1 Tabular Methods

Tabular modelling methods, such as Dyna-Q [87] and its probabilistic adaptation presented in section 2.3 use labelled states to model the environment. The relative frequency of each successor state is used to provide an empirical probability of the results of each action in each state. This method provides a useful benchmark because it has the modelling power to create a perfect model of a Markov environment given sufficient training data. The method suffers from the "curse of dimensionality" [5], because the table size is exponential to the number of perceptual features in the environment.

8.1.2 MSDD

The MSDD algorithm, defined in section 3.1.2, has been used to learn a P-SPO set in previous research by the author [16]. MSDD learns individual rules, which are combined to form P-SPOs using the ASDD supplementary algorithms (defined in chapter 6).

8.1.3 Learning a Policy from the Model

A tabular policy can be generated from a world model by using standard reinforcement learning or dynamic programming techniques (section 3.2). These methods have been evaluated by the author in [17]. Reinforcement learning methods map state-action pairs to values, while dynamic programming methods can map state-action pairs to values, or can map states to values and use the model to evaluate the highest valued action. The method is agnostic to the type of model used, with the result that a direct comparison of the quality of policy produced by each model can be made.

Each test used a fixed model, and a learning rate of 1.0 can, therefore, be used for the policy learning algorithm if full-backups are performed (as described in section 3.2.2). This is equivalent to dynamic programming using a post-decision state variable representation, as described by Powel [73]. Discount rates for future reward were set at 0.9 for all tests.

8.1.4 Error Measure

The error measure provides an indication of the number of states (or equivalently percepts) missing from the successor states generated by the model, the number of additional states generated by the model and the differences in probabilities indicated by the model.

The measure is that defined in section 6.6 for evaluating the supremacy between P-SPO operators:

- *For each non-matching outcome:* add +0.5. A non-matching outcome is one that is present in the combined rule but not in the original rule, or present in the original rule but has probability zero in the combined rule.
- *For each matching outcome:* add the absolute difference between the empirical probability for the outcome, and that for the outcome generated by the perfect model.

The error measure requires that we have an accurate model available with which to compare the states and probabilities generated by the models to be tested. These *comparison environment models* were generated from very large data sets, using the tabular method, by running the test model for 3 million moves. It should be noted that these models will contain inaccuracies, due to unreliable empirical probability generation for rarely visited states, but these will be minor in models of these size used in the tests.

Algorithm 8-1 was used for error measure generation. The algorithm cycles through all the sate-action pairs contained in the comparison model and returns the sum of the *errorMeasure* evaluation of the difference between the generated states and the comparison model states.

```
findModelError(M)
   initialise C with comparison model
   E = 0;
   //cycle through all percept action pairs in C
   for all ({p,a}∈C) {
      //find the outputs and associated probabilities
      //for comparison model C and model M
      distributionModel_C:p×a→{O_C,P_C};
      distributionModel_M:p×a→{O_M,P_M};
      //add error measure for incorrect outputs or probabilities
      E += errorMeasure({O_C,P_C}, {O_M,P_M});
   }
   return E;
```

Algorithm 8-1: findModelError. M=model to be compared. The function returns the error measure for the model to be compared against an exhaustive tabular model for the same environment.

8.1.5 Time Taken Comparison

Performance timings of the algorithms were taken on a 2 GHz Intel Centrino processor with 2GB of RAM. A comparison was made of the time taken by each model learner for the given data set. All comparisons are for the same input data sets of perceptual data items (PDIs).

8.2 Results: Slippery Gripper with Additional Dependencies

The slippery gripper environment with additional dependencies has:

- States: 20
- State-action pairs: 80 (20 states times 4 actions available in the environment)
- State-action following states: 148

8.2.1 Model Accuracy

Table 8-1 shows the error measure for the model generated by each of the model learning methods. The error for ASDDs is equal to that for ASDD because the same training data set was used for all experiments and the ASDDs method is essentially the same algorithm with an optimised counting method.

Table 8-1: Error measure of generated states generated from rules learned from data collected over 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

	100	1000	5000	10000	20000	50000	100000
Tabular	56.27	24.13	7.84	7.04	3.53	1.18	1.55
MSDD	66.40	17.85	6.76	5.41	4.25	0.77	0.30
ASDD	75.90	17.06	4.95	3.57	2.32	0.73	0.27
ASDDs	75.90	17.06	4.95	3.57	2.32	0.73	0.27

The error measure values show that P-SPOs are capable of learning an improved model in comparison to a tabular method for all but the most limited training data. It is possible to learn a perfect model of the environment using P-SPOs because the domain has independent output variables. The table shows that the ASDD and MSDD algorithms are capable of learning this model from data.

Figure 8.1 shows the graph of these data values. ASDD performs poorly on the initial data set but, with a training data size of 1000 or more PDIs, the ASDD algorithm learns the most accurate model. The main difference between the rule learning capabilities of MSDD and ASDD is the *aprioriFilte*r function in ASDD, which removes rules with low significance early in the process. This has the effect of removing rules that would over-fit the data in this domain.

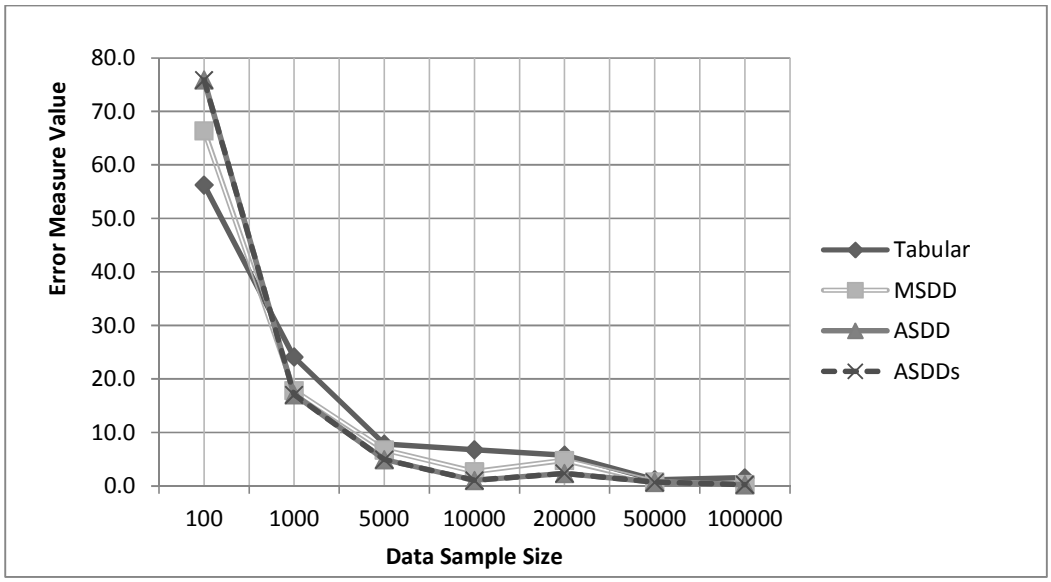


Figure 8.1: Graph of error measure of generated states generated from models generated from data collected over 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

Table 8-2 shows the extra states and missing states generated by each of the models. The numbers are reflected in the error measure because each extra/missing state adds 0.5 to the

error measure. The tabular method cannot generate additional states because this would mean generating states that cannot be reached.

The states missed by the tabular methods for a data set of 100,000 random moves are those for the action *new* with initial percept:

{*painted(true), clean(true), dry(false), holding(true), reward(none)}*

The percept is rarely seen with random moves because the gripper always becomes dirty if the *paint action* is used while holding the block, and often becomes dirty if not holding the block. The percept can only occur in the unlikely event that the block was painted while not held, and then was picked up while the gripper was dry (also unlikely because a *dryer* action is required after the initial *paint*). This percept and action combination has not been observed and the model is not able to generate any successor states. The correct successors are:

{*painted(false), clean(true), dry(false), holding(false), reward(pos)} Pr: (0.75)*

{*painted(false), clean(true), dry(true), holding(false), reward(pos)} Pr: (0.25)*

The P-SPO based methods are prone to generating additional states initially, but will gradually learn a completely correct model because the environment's characteristics allow it to be modelled by operators of this type.

An example of incorrect states generated by operators with insufficient data from the 100 trial ASDD rule set is for the action *paint* with initial percept:

{*painted(true), clean(false), dry(false), holding(true), reward(none)*}.

The correct successor percept is for the state to remain unchanged (because the block was already painted and the gripper was wet). The P-SPO set, however, generated two states, with the additional one being:

{*painted(true), clean(false),* ***dry(true)****, holding(true), reward(none)*}.

This was caused by the following ASDD generated rule:

$$\{\}: clean(true), rew(pos) \rightarrow \begin{Bmatrix} 0.8: dry(false) \\ 0.2: dry(true) \end{Bmatrix}.$$

Table 8-2: Missing states vs. extra states generated by each model. The first number in each cell is the number of states missing from the model and the second number indicates extra states generated by the model.

	100	1000	5000	10000	20000	50000	100000
Tabular	101-0	35-0	11-0	10-0	10-0	1-0	2-0
ASDD	75-67	22-0	7-0	1-0	4-0	1-0	0-0
MSDD	72-48	22-0	7-0	2-0	4-0	1-0	0-0

An anomaly in this table is that ASDD learns a more accurate rule set than MSDD with a model generated from 10,000 data items. The error is for the action *new* with initial percept:

{*painted(false), clean(false), dry(false), holding(true), reward(none)*} *Pr(0.7)*

The correct successors for the percept are:

{*painted(false), clean(true),* ***dry(false),*** *holding(false), reward(neg)*} *Pr(0.7)*

{*painted(false), clean(true), dry(true), holding(false), reward(neg)*} *Pr(0.3)*

The MSDD-based model missed the first of these states (the one with *dry(false)*), because it used an over specific P-SPO that predicts *dry(true)* with probability 1.0:

$$new: painted(false), clean(false), dry(false), holding(true) \rightarrow \{1.0 : dry(true)\}$$

This operator had been filtered by the *aprioriFilter* function in ASDD, leaving the simple (and correct) rule to have precedence:

$$new: \{\} \rightarrow \begin{Bmatrix} 0.7 : dry(false) \\ 0.3 : dry(true) \end{Bmatrix}$$

8.2.2 Speed of P-SPO Set Learning

Table 8-3 shows the time taken by each model learning algorithm. The tabular rule learning method makes a single pass through the data and is therefore very fast in comparison to both rule learning methods. MSDD, ASDD and ASDDs are each approximately linear in time to the size of the data set. Intuitively, this is expected because each method has to perform a count of the number of times the rules match the data set. MSDD has to perform the count more often than ASDD, while ASDDs has a faster counting mechanism than ASDD, requiring only one full pass through the PDI database.

Table 8-3: Time taken (in seconds) to learn a P-SPO set or tabular model with data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

	100	1000	5000	10000	20000	50000	100000
Tabular	0.02	0.02	0.02	0.02	0.03	0.08	0.16
MSDD	1.03	13.09	63.49	181.38	430.70	1018.92	1957.41
ASDD	1.76	4.04	8.74	14.27	29.25	61.12	152.05
ASDDs	1.06	1.67	2.24	4.17	11.58	28.02	72.51

Figure 8.2 shows the graph of these data values for the operator learning methods. The graph is shown with a scale of Log10 time in seconds. The size of the training set is an approximately logarithmic scale.

It is clear from the graph that the learning time taken for all rule learning methods is approximately proportional to the training data size. ASDD is approximately 13 times faster than MSDD and ASDDs is approximately double the speed of ASDD for larger training sets.

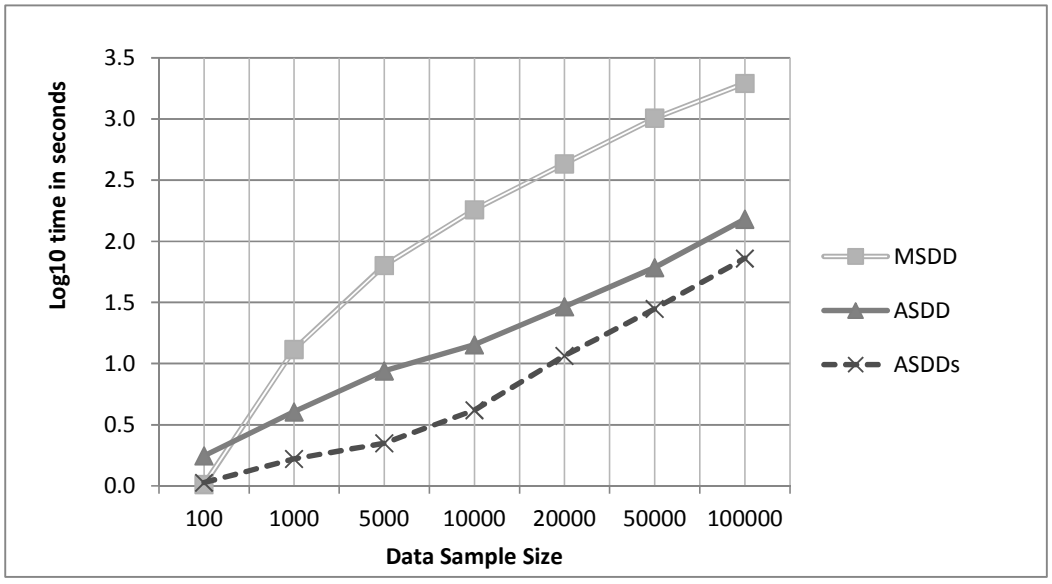


Figure 8.2: Graph of time taken (in seconds) to learn a P-SPO set or tabular model with data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

8.2.3 Reward Gathered by a Policy Learned from the Data

Rewards are +1 for a painted block delivered, and -10 for an unpainted block delivered. Policies are learned using dynamic programming set to 10,000 iterations. Dynamic programming builds a table of state to value (rather than state-action to value).

- Discount level, γ, is set to 0.9.

- There is no learning rate required (equivalent to learning rate 1.0) because this is a model based technique with full backups.
- The policy is taken to be the action with the highest expected future rewards (see section 3.2.2 for definitions).

Table 8-4: Reward gathered after following a policy derived from a model learned from data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

	100	1000	5000	10000	20000	50000	100000
Tabular	6798	26136	26095	26139	26076	26166	26097
MSDD	26099	26050	26133	26112	26120	26079	26109
ASDD	26121	26129	26078	26127	26072	26120	26123

Figure 8.3 shows the graph of these data values. The graph shows that P-SPO sets are able to learn a perfect policy with a very small amount of experience, using the same training data for which the tabular method is not able to perform to this level. The small differences in "perfect" policies are due to random experimental variation.

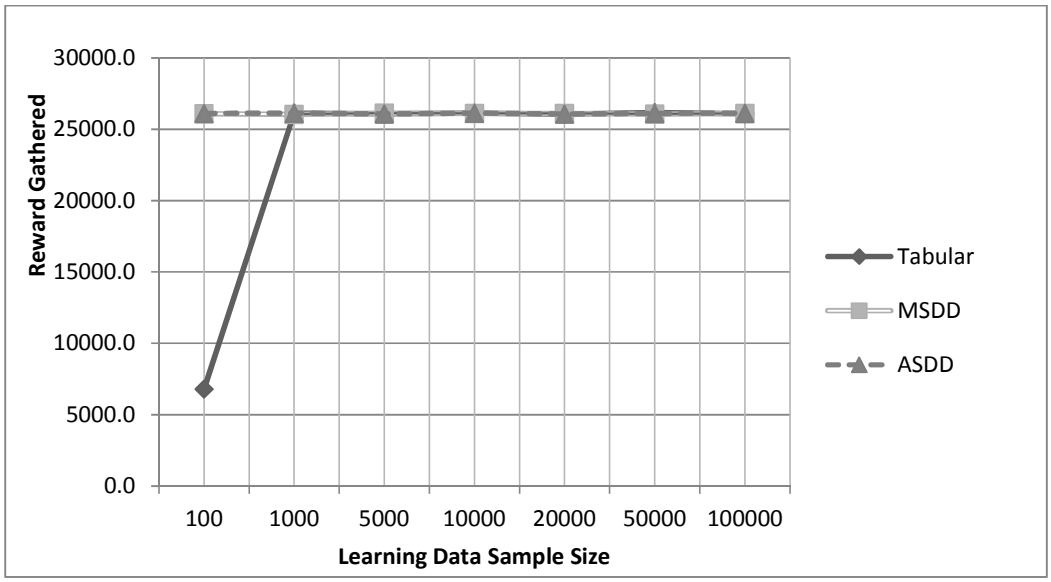


Figure 8.3: Graph of reward gathered after following a policy derived from a model learned from data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

8.2.4 Goals Achieved vs. Disaster States Encountered

Table 8-5 shows the number of painted blocks delivered and the number of unpainted blocks delivered. The tabular method performs poorly when limited experience is available, because it must take a random action if it has not encountered the state before. At this level of experience, some of the states are not visited. The model contains only 20 states, but some of

them are rarely visited, and it is therefore unlikely that they will be encountered with limited experience. The rule based model is able to model the effects of actions at this limited experience level and can, therefore, always avoid the disaster states.

Table 8-5: Goals achieved vs. disaster states encountered after following a policy derived from a model learned from data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves. The first number in each cell is goals achieved. The second number is disaster states encountered.

	100	1000	5000	10000	20000	50000	100000
Tabular	24348-1755	26136-0	26095-0	26139-0	26076-0	26166-0	26097-0
ASDD	26099-0	26050-0	26133-0	26112-0	26120-0	26079-0	26109-0
MSDD	26121-0	26129-0	26078-0	26127-0	26072-0	26120-0	26123-0

8.2.5 Comparison of Learned vs. Actual P-SPO Set

Figure 8.4 show the learned P-SPO set for 100,000 data items. The rule set shown includes only those rules that are used in successor generation (those that have precedence in at least one state-action situation). Environment operators are not shown for clarity. The operator set is an almost completely accurate representation of the P-SPO set for the environment. The errors shown in the graph in section 8.2.1 are, in some cases, reflecting small errors in the *comparison environment model* (the tabular model generated from a data-set of 3 million PDIs), demonstrating the improved modelling power possible with the algorithm.

Two of the rules below have slightly incorrect probability estimates due to differences in the base probabilities in the sample (shown in bold).

$$paint : holding(true) \rightarrow \{1.0 : painted(true)\}$$

$$paint : holding(true) \rightarrow \{1.0 : clean(false)\}$$

$$paint : holding(false) \rightarrow \begin{Bmatrix} 0.8 : clean(true) \\ 0.2 : clean(false) \end{Bmatrix}$$

$$paint : \begin{matrix} painted(false), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.1 : painted(true) \\ 0.9 : painted(false) \end{Bmatrix}$$

$$paint : \begin{matrix} clean(true), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.2 : clean(true) \\ 0.8 : clean(false) \end{Bmatrix}$$

$$paint : holding(true) \rightarrow \{1.0 : clean(false)\}$$

$$dryer : dry(false) \rightarrow \begin{Bmatrix} 0.9 : dry(true) \\ 0.1 : dry(false) \end{Bmatrix}$$

$$\textbf{\textit{pickup :}} \begin{matrix} \textbf{\textit{dry(true), painted(false),}} \\ \textbf{\textit{holding(false)}} \end{matrix} \rightarrow \begin{Bmatrix} \textbf{\textit{0.93 : holding(true)}} \\ \textbf{\textit{0.07 : holding(false)}} \end{Bmatrix}$$

$$pickup : \begin{matrix} dry(true), painted(true), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.75 : holding(true) \\ 0.25 : holding(false) \end{Bmatrix}$$

$$pickup : \begin{matrix} dry(false), painted(false), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.15 : holding(true) \\ 0.85 : holding(false) \end{Bmatrix}$$

$$pickup : \begin{matrix} dry(false), painted(true), \\ holding(false) \end{matrix} \rightarrow \begin{Bmatrix} 0.05 : holding(true) \\ 0.95 : holding(false) \end{Bmatrix}$$

$$\textbf{\textit{pickup :}} \begin{matrix} \textbf{\textit{painted(true),}} \\ \textbf{\textit{clean(true)}} \end{matrix} \rightarrow \begin{Bmatrix} \textbf{\textit{0.17 : clean(true)}} \\ \textbf{\textit{0.83 : clean(false)}} \end{Bmatrix}$$

$$new : painted(false) \rightarrow \{1.0 : reward(neg)\}$$

$$new : painted(true) \rightarrow \{1.0 : reward(pos)\}$$

$$new : \{\} \rightarrow \{1.0 : holding(false)\}$$

$$new : \{\} \rightarrow \{1.0 : painted(false)\}$$

$$\{\} : \{\} \rightarrow \begin{Bmatrix} 0.3 : dry(true) \\ 0.7 : dry(false) \end{Bmatrix}$$

$$environment : \{\} \rightarrow \{1.0 : reward(none)\}$$

Figure 8.4: P-SPOs generated by ASDD with a training data set of 100,000 for the "slippery gripper" domain.

There is a difference in the *new* action's effect on the *dry* state of the gripper. The rule should have an output:

$$new : \{\} \rightarrow \begin{Bmatrix} 0.3 : dry(true) \\ 0.7 : dry(false) \end{Bmatrix}$$

The action of drying a wet gripper, coincidentally, has exactly the same probability as the background probability of the gripper being dry (shown with a rule with no action or conditions).

$$\{\}:\{\} \rightarrow \begin{Bmatrix} 0.3:dry(true) \\ 0.7:dry(false) \end{Bmatrix}$$

This is a more general rule, and the background probability therefore has precedence over the dryer action. This does not cause any damage to the model, because any operators that have a significantly different output probability to the base chance will have precedence.

8.2.6 Results Discussion

The P-SPO based model is extremely effective in the slippery gripper environment. With a small amount of environmental experience, operators can be learned which have a low error measure and from which an optimal policy can be derived.

Both MSDD and ASDD are effective at operator learning in this test-case, with the ASDD based operators showing improved performance by eliminating over-specific rules, and being learned in a reduced amount of the time. The set-based optimisation shows useful speed improvements at this level.

The rule sets learned by MSDD and ASDD methods accurately capture the properties of the original rule set, given sufficient experience, and are able to capture a low fidelity model of the environment given reduced training data.

8.3 Results: Predator Prey Environment

An implementation of the ASDD algorithm has been tested against the MSDD algorithm for speed of model learning, accuracy of the model, and the policy achieved by the agent.

The predator-prey environment has:

- States: 42
- State-action pairs: 168 (42 × 4 actions available in the environment)
- State-action following states: 732

8.3.1 Model Accuracy

Table 8-6 gives the error measure of the state generation ability of ASDD, MSDD and a state map against an empirical measure of the state transition probabilities taken from a state map of 3 million random moves (a “perfect” state map).

The performance of both rule-learning methods is poor against a state map generated from the same number of trials, with the exception of the case where there is a limited amount of data. The performance of the rule sets generated by ASDD and MSDD are, however, approximately equal in model performance.

The error for ASDDs is, again, exactly the same as ASDD because the ASDDs method is essentially the same algorithm with an optimised counting method.

Table 8-6: Error measure of generated states generated from rules learned from data collected over 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

	100	1000	5000	10000	20000	50000	100000
Tabular	352.74	267.83	49.95	18.24	4.88	3.94	1.55
MSDD	389.86	260.79	123.58	141.98	84.62	66.60	43.25
ASDD	391.50	237.93	175.49	204.25	130.53	103.34	42.03
ASDDs	391.50	237.93	175.49	204.25	130.53	103.34	42.03

Figure 8.5 shows the graph of these data values. The graph shows that a large amount of experience is required by the P-SPO learning algorithms to achieve an accurate model. This is a natural consequence of the properties of the environment. Most output variables in the environment are interdependent. If, for example, the predator moves onto the prey's square, then the prey must be present in the predator's successor percept. It will be in only one square, and all other squares will therefore be empty (or contain a wall). The only squares which are not dependent in this way are the walls. If a predator moves along, or into, a wall, the wall will remain in the square with probability 1.0.

A further issue is that output probabilities and dependencies require knowledge of several inputs, with, in some cases, dependencies hidden for rules with multiple conditions (similar to the XOR problem mentioned in chapter 6).

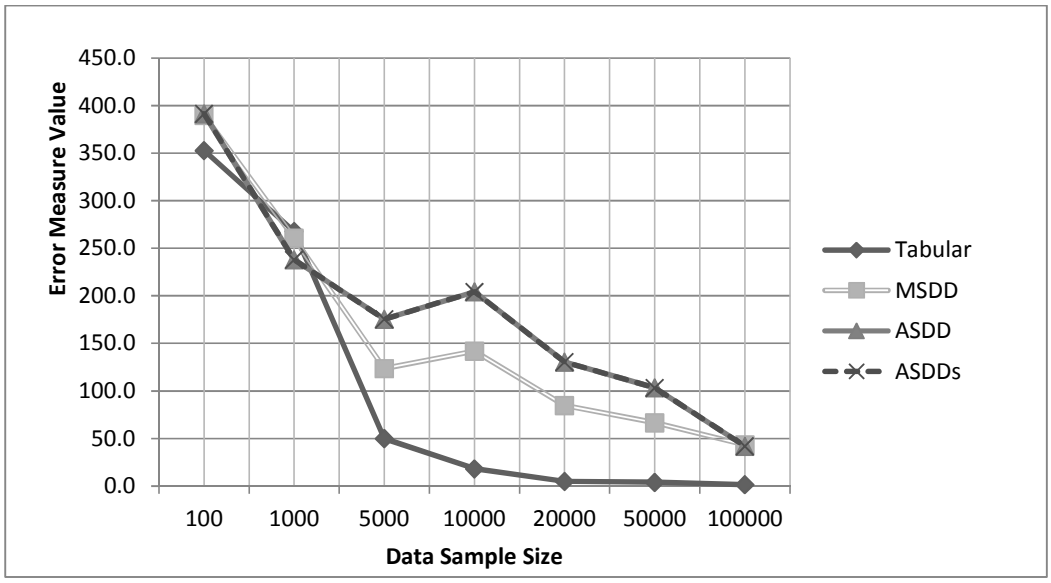


Figure 8.5: Graph of error measure of generated states generated from models generated from data collected over 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

Table 8-7 shows the extra states and missing states generated by each of the models. Both the ASDD and MSDD algorithms perform poorly on this learning task. The accuracy of the rules improves as the data sample size increases, demonstrating that it is possible to learn reasonable rules in this form, but the environment has multiple dependencies between inputs as well as outputs, making it difficult to make effective rule-based generalisations. The ASDD algorithm's *aprioriFilter* step compounds this issue because important dependencies are filtered at an early stage, with required conditions often not significant until multiple state's conditions have been added. The tabular model is effective in this environment because it equates to rules with multiple dependencies between outcomes, and for which all inputs are present in the conditions.

Table 8-7: Missing states vs. extra states generated by each model. The first number in each cell is the number of states missing from the model and the second number indicates extra states generated by the model.

	100	1000	5000	10000	20000	50000	100000
Tabular	671-0	432-0	91-0	28-0	0-0	0-0	0-0
ASDD	560-156	358-86	61-171	270-111	155-4	122-68	0-79
MSDD	558-169	301-128	110-225	235-32	209-37	98-30	0-82

8.3.2 Speed of P-SPO Set Learning

Table 8-8 shows the time taken by each model learning algorithm. The tabular rule learning method makes a single pass through the data and is therefore very fast in comparison to rule learning methods. MSDD, ASDD and ASDDs are each approximately linear in time to the size of the data set, as was the case with the "slippery gripper" environment. This makes intuitive sense because each method has to perform a count of the number of times the rules match the data set.

Table 8-8: Time taken (in seconds) to learn a P-SPO set or tabular model with data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

	100	1000	5000	10000	20000	50000	100000
Tabular	0.02	0.02	0.02	0.02	0.03	0.09	0.16
MSDD	1.58	39.03	218.88	390.70	884.57	2300.81	4558.42
ASDD	2.56	12.78	22.21	32.20	54.23	133.29	322.80
ASDDs	1.54	5.27	5.69	9.41	21.47	61.11	153.93

Figure 8.6 shows the graph of these data values. The tabular learning method is very fast and therefore cannot be represented at the scale of the graph. The graph shows that there is a slight overhead for ASDD on small training sets, after which ASDD and ASDDs show dramatic increases in speed over MSDD.

ASDD is approximately 15 times faster than MSDD at learning an operator set, with variations in the initial speeds. ASDDs is approximately two times faster than ASDD with, again, variations in the initial speeds.

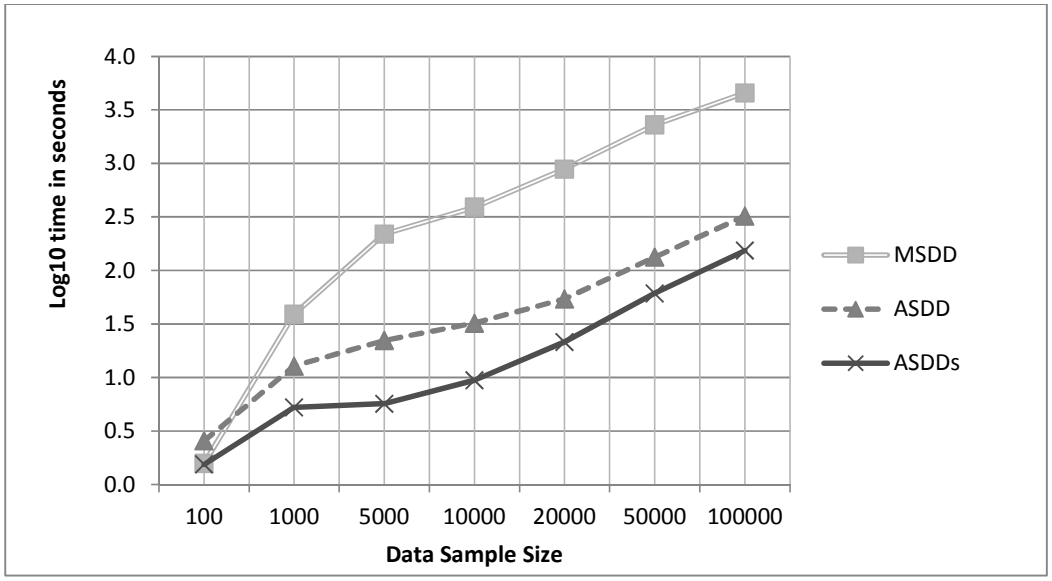


Figure 8.6: Graph of time taken (in seconds) to learn a P-SPO set or tabular model with data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

The time taken to learn a full P-SPO set at each data sample size and by each algorithm is approximately double that taken to learn the "slippery gripper" environment P-SPO sets. No firm conclusions can be drawn from this, but it is interesting to note that the two environments have the same number of perceptual features, but the rules generated for the predator-prey environment are more complex, defining an environment with an increased number of possible states and greater dependency between variables.

8.3.3 Reward Gathered by a Policy Learned from the Data

Rewards were set at +1 for each time-step in which the predator was on-top of the prey at the start of its move and 0 otherwise. Policies are learned using dynamic programming set to 10,000 iterations.

- Discount level, γ, is set to 0.9.
- There is no learning rate required (equivalent to learning rate 1.0) because this is a model based technique with full backups.
- The policy is taken to be the action with the highest expected future rewards.

The test was run over 100,000 iterations, with the maximum achievable reward being received if the predator follows a policy of following the prey every move. The predator will then receive a reward each time the prey moves onto a wall (therefore staying in the same square). This happens approximately 1 in 4 moves on average (probability 0.25) giving a maximum expected reward of 25,000. A minimum expected reward policy is achieved by taking random moves, in which case the predator will be on-top of the prey 1 in 16 moves, giving a reward of 6,250.

Table 8-9: Reward gathered after following a policy derived from a model learned from data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

	100	1000	5000	10000	20000	50000	100000
Tabular	7939.0	9466.0	17137.0	11078.0	18440.0	20473.0	21921.0
MSDD	8126.0	9470.0	9257.0	10117.0	9040.0	6137.0	11467.0
ASDD	7041.0	6181.0	8114.0	9407.0	11355.0	11813.0	15956.0

Figure 8.7 shows the graph of these data values. The graph shows that the tabular method offers the most successful model for policy formation. This is expected because the environment contains multiple dependencies between outcomes and is challenging to model via P-SPOs. Neither ASDD nor ASDD are able to learn a perfect model of the environment which means they are also unable to learn an optimal policy.

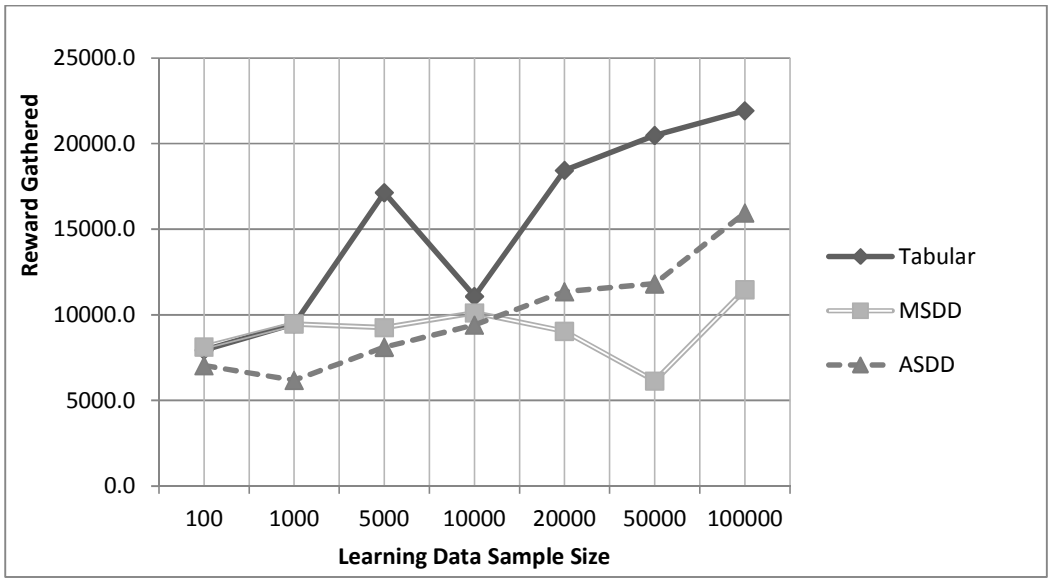


Figure 8.7: Graph of reward gathered after following a policy derived from a model learned from data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

A point of interest in the reward graph is that ASDD is able to lean a more effective policy from the rule set learned, despite the error measure showing that MSDD's rules are more accurate. This is caused by over-specific rules relating to the position of the prey agent in response to a move. This is the most important aspect of the model, so small changes can cause problems.

Another key point is that MSDD's performance at the 50,000 data level drops to worse than that of a random policy. On examination of the agent's actions under this policy, it was found that the agent moved to the north east corner of the map and stayed in that square, irrespective of the prey agent's moves.

Examining the agent's action under the policy learned using the ASDD rules, acquired from 100,000 PDIs, shows that the agent moves on-top of the prey in most situations, but will occasionally move into a wall, rather than chasing the prey as it moves into open space, causing the prey to go out of view and be lost for several moves. This policy is optimal given the agent's incomplete world model, because the agent's model does not capture the fact that moving onto the prey keeps it in the percept (see following section).

Overall, this is a challenging environment in terms of both modelling and policy formation for a P-SPO-based system, with the assumption of independent output variables, but the system is still able to learn an effective, if not optimal, policy.

8.3.4 Inspection of Learned P-SPO Set

The predator prey environment is a simulation environment and is not based on a P-SPO set. It is not therefore possible to compare the acquired operator set with the actual set. The full set is also large, due to the difficulty in representing dependent output variables with independent outcomes. Figure 8.8 gives a sample of the operators learned from 50,000 PDIs to give an example of the type of rules the system learns.

Given an action *move(south)* and an initial percept:

$$see(north, wall), see(east, wall), see(south, agent), see(west, empty), see(under, empty)$$

The matching operators after filtering by precedence are:

$$move(south) : see(south, agent) \rightarrow \left\{ \begin{array}{l} 0.24 : see(north, agent) \\ 0.76 : see(north, empty) \end{array} \right\}$$

$$move(south) : see(south, agent), see(west, wall) \rightarrow \left\{ \begin{array}{l} 0.23 : see(east, agent) \\ 0.77 : see(east, empty) \end{array} \right\}$$

$$move(south) : see(north, wall), see(south, agent) \rightarrow \left\{ \begin{array}{l} 0.24 : see(south, agent) \\ 0.76 : see(south, empty) \end{array} \right\}$$

$$move(south) : see(west, wall) \rightarrow \{1.0 : see(west, wall)\}$$

$$move(south) : \begin{array}{c} see(north, wall), see(south, agent), \\ see(west, wall) \end{array} \rightarrow \left\{ \begin{array}{l} 0.25 : see(under, agent) \\ 0.75 : see(under, empty) \end{array} \right\}$$

Figure 8.8: Matching P-SPOs for the move(south) action in the predator-prey environment for a set of operators acquired from 50,000 PDIs experiences.

The above operators are a compact representation of the probabilities for each outcome and contain no unnecessary conditions. The second operator, for example, gives the probability of seeing the prey-agent to the east if the predator has moved south, sees the prey to the south, and sees a wall to the west. The square to the east must be empty with these conditions, and the predator is vacating the square which will become the square to the north next. The prey takes a simultaneous move, during which it may move into the vacant square to the east (randomly taking a move in one of four directions), which is accurately captured by the conditions (with a small sampling error).

The first operator has only one condition, but accurately finds the probability that the prey will move into the square vacated by the predator. No further conditions are required because this square is certain to be empty and the prey can, therefore, move into it with probability 0.25. The empirical probability of 0.24 given by the rule matches this closely because all available evidence can contribute to the estimate.

The states generated by the rules above, after removal of *invalid* states (e.g. those containing more than one agent) are:

$see(north,empty), see(east,empty), see(south,empty), see(west,wall), see(under,empty)$ Pr(0.33)
$see(north,empty), see(east,empty), see(south,empty), see(west,wall), see(under,agent)$ Pr(0.11)
$see(north,empty), see(east,empty), see(south,agent), see(west,wall), see(under,empty)$ Pr(0.11)
$see(north,empty), see(east,agent), see(south,empty), see(west,wall), see(under,empty)$ Pr(0.08)
$see(north,agent), see(east,empty), see(south,empty), see(west,wall), see(under,empty)$ Pr(0.11)

Normalising the above state probabilities (summing probabilities and dividing each by the sum so that state probabilities sum to 1), gives:

***see(north,empty), see(east,empty), see(south,empty), see(west,wall), see(under,empty)*Pr(0.45)**
$see(north,empty), see(east,empty), see(south,empty), see(west,wall), see(under,agent)$ Pr(0.15)
$see(north,empty), see(east,empty), see(south,agent), see(west,wall), see(under,empty)$ Pr(0.15)
$see(north,empty), see(east,agent), see(south,empty), see(west,wall), see(under,empty)$ Pr(0.10)
$see(north,agent), see(east,empty), see(south,empty), see(west,wall), see(under,empty)$ Pr(0.15)

The first of these output states (shown in bold) highlights the issue with treating non-independent variables as if they were independent (discussed in chapter 6). It is not possible for the prey agent to be outside the percept of the predator if it moved onto the prey square. Treating the outputs as independent, however, the state with no agent present is predicted to be the most likely successor state. Methods of resolving this are addressed in section 11.3.

It should be noted that there are no environment operators in the predator prey environment because all actions can influence all perceptual features.

8.3.5 Results Discussion

The key element missing from the predator prey model acquired using the limited P-SPOs acquired by ASDD and MSDD is that a predator moving onto the prey's square cannot result in the prey moving outside the predator's view. The model produces multiple possible outputs with no prey present.

The predator prey environment used was challenging for both MSDD and ASDD rule learning algorithms in that it contains exogenous events and dependencies between outputs. The current implementation of the rule learning algorithms does not include dependencies between outputs and is therefore unable to learn an accurate model in this circumstance. This results in poor performance against a state map as the model data size becomes larger. Model accuracy results are varied for ASDD because the *aprioriFilter* step can remove important rule elements from later consideration.

The implementation of the ASDD algorithm (section 5.8) assumes independent outputs and therefore generates a set of rules with only one perceptual feature in the outcome set of each operator. This form of the algorithm can substantially reduce the search space of possible operators for environments with independent outcomes, at the cost of reduced performance in environments with dependent outcomes, but can also reduce over-fitting of data by removing over-specific rules.

If dependencies between outcomes have not been correctly modelled by the P-SPOs, impossible successor percepts must be removed by the use of constraints (section 5.4.5). In the predator-prey scenario, the operators may generate a percept with two agents when there is only one agent in the world. If we do not use these constraints, the erroneous generated states will propagate (e.g. predator agents, three walls etc.), and the model becomes meaningless, because it is too far detached from the real world states. Currently the system removes impossible states using the *invalid* function to check that each generated state contains only one agent, and does not contain walls opposite each other.

After elimination of illegal states, the probabilities of remaining states are normalised by dividing the probability of each state by the total probability of all generated states to give the final states.

8.4 Summary

This chapter has presented performance results of the ASDD and ASDDs algorithms against a standard tabular model and the MSDD algorithm for speed of operator set learning, accuracy of the operator sets, and performance of policies derived from the operator sets.

ASDD and ASDDs were shown to be approximately 15 and 30 times faster at learning P-SPO sets than MSDD respectively, with variations for small PDI training data samples. The speed of all algorithms was approximately linear to the size of the training data set. Differences between the time taken to learn a slippery gripper P-SPO set versus time taken to learn a predator-prey P-SPO set show a relation between complexity of operators required to model the state-space and time taken for learning to complete. This result requires further experimentation for effective conclusions to be drawn.

The P-SPO set learned by ASDD for the “slippery gripper” environment produced an accurate environment model with limited experience and was able learn an optimal policy with only a small sample of data from the environment. The model accuracy was slightly improved over that learned by MSDD because the *aprioriFilter* step was able to eliminate over-specific operators which caused over-fitting of the data in the MSDD model.

The P-SPO set learned by ASDD for the predator-prey environment demonstrated the effectiveness of the operator learning method in extracting a minimal rule set from data, but also demonstrated the issues with attempting to model environments with large dependencies between output variables using a system that does not have this modelling capability.

The error-measure based accuracy of the model learned by ASDD was, naturally, poor in comparison to that learned by a tabular method, but was effective in comparison to the MSDD based model. The policy learned using the ASDD model was, however, superior to that learned by MSDD. The reasons for this are not clear from the data, but investigation of the policy employed by the agent shows that it has over-fit the data and therefore "expects" the prey to pick the same moves it used in training. This can lead to a policy of, for example, staying in one corner of the map.

9. Rule Value Reinforcement Learning (RVRL)

This chapter presents Rule Value Reinforcement Learning (RVRL), an algorithm which uses an approximate dynamic programming based technique [87] to attach values to a parallel-stochastic planning operator model of an environment. The operators can then be used to compactly represent a policy for an agent, reducing the need for the exponentially large value map required by standard dynamic programming methods.

Rule Value Reinforcement Learning (RVRL) was first introduced in [16], and was presented within an operator learning framework in [15].

The RVRL algorithm iteratively updates values attached to P-SPOs. The principle is that structure captured in the rules can be used to learn an approximate policy directly. The resulting value attached to each operator represents a utility estimate for taking an action if the conditions of the operator are present in the agent's current percept. A set of P-SPOs are used in parallel when generating successor percepts or states. This means that RVRL must define the contribution between the values of each operator to the overall value of taking an action under the current conditions. This chapter defines the original RVRL approach (an average over operator values), and alternative algorithms (state aggregation techniques based on approximate dynamic programming [73]).

The principles behind the RVRL algorithm are:

- P-SPO learning algorithms capture structure in the environment from the perspective of actions within it.
- The utility and reward associated with environment states is related to the environment structure.
- Associating utility with the operators will capture the utility structure, allowing useful generalisations to be made between states (state aggregations) and avoiding the need for an exponentially large state-action utility map (with respect to the number of features and actions in the environment).

It should be noted that there is currently no known algorithm for extracting useful state aggregations from data. Finding appropriate aggregation functions is, according to Sanner, "more of an art than a science" [79], while Powell points out that an appropriate state aggregation function for an environment requires extensive domain knowledge and, when discovered, constitutes a patentable result [73]. The automated state aggregation method presented here constitutes a novel contribution to the field of approximate dynamic programming.

The following sections: introduce the process of attaching values to P-SPOs; define the *average rule value* update function; and show how improved values can be generated iteratively. The process of following a policy generated from the operator values is given, and finally an alternative *state aggregation* update function (based on approximate dynamic programming techniques) is defined.

9.1 Attaching Utility to P-SPOs

The RVRL algorithm attaches a utility to each operator in the P-SPO model. The simple coin flipping agent, introduced in chapter 2, will be used as an example to demonstrate the basic concepts.

The example environment consists of:

- A single coin.
- The agent can either *flip* the coin or *doNothing*.
- The reward for the coin *showing heads* is 1 and *showing tails* is 0.

If the environment is episodic and the result of an action is a terminating state, then the utility of the *flip* action can be calculated very simply by using dynamic programming, which sums the immediate reward (1 if the result is *heads,* and 0 if it is *tails*) multiplied by the probability of each (giving 1×0.5 + 0×0.5) plus the discounted rewards of all future actions from the following states (0, because there are no future actions).

The *doNothing* action is deterministic, and does not change the state. Its utility is, therefore, simply the reward received in the current state.

A full P-SPO set for the coin-flipping agent is shown below:

$$\begin{aligned} &flip:\{\} \rightarrow \begin{Bmatrix} 0.5:showing(heads) \\ 0.5:showing(tails) \end{Bmatrix} U(0.5) \\ &doNothing:showing(heads) \rightarrow \{1.0:showing(heads)\} U(1.0) \\ &doNothing:showing(tails) \rightarrow \{1.0:showingTails(tails)\} U(0.0) \end{aligned} \tag{9.1}$$

If we have an initial state of: *{showing(tails)}*, then the operators that match this state are:

- *doNothing* (with the context *showing(tails)*).
- *flip (*with the empty context, *{}*).

The utility of the *doNothing* action in the *{showing(tails)}* state is 0.0, while the *flip* action has utility 0.5. The agent should, therefore, take the action with the highest utility (0.5) and flip the coin. If the environment state was *showing(heads),* then the best option would be *doNothing* (with utility 1.0).

If a full value-map is known, then the utilities can be derived from the value map of the states:

$$showing(heads) : U(1.0)$$
$$showing(tails) : U(0.0)$$

The utility of the *doNothing* action is the utility of the resulting state after the action, leading to the utilities of the two *doNothing* operators (9.1). The utility of the *flip* action is the utility of the resulting state after the action, which in this case is stochastic, so the probability of reaching the resulting state multiplied by the utility of the state is used, giving ($0.5 \times 1.0 + 0.5 \times 0.0 = 0.5$).

The "curse of dimensionality" [5] means that it is often not possible to derive a full state-value map for an environment with a large number of features. The RVRL algorithm provides a method for attaching utility to operators without the need to build the value-map. The RVRL algorithm iteratively improves estimates of operator utilities using the previous estimate attached to the operators matching the successor states.

9.1.1 Two Coin Example with Environment Operators

The coin flipping example can be extended to two coins to demonstrate the intended output of the algorithm as an aggregation of values over states. The example also includes *environment* operators (introduced in chapter 5).

The example environment consists of:

- Two coins.
- The agent can either *flip* one of the coins or *doNothing*.
- The reward for both coins *showing heads* is 1.0.
- The reward for both coins *showing tails* is 0.0.

The environment is episodic, with the terminating states being: both coins show heads; or both coins show tails. The agent continues to take actions until a terminating state is reached.

An initial state for an environment of this type is:

$$\{showing(penny, tails), showing(pound, tails)\}$$

The extension to two coins requires the addition of an *environment* operator to model the evolution of the parts of the environment that are not affected by an action.

As defined in section 5.7, the *environment* operator has the form of an action, *environment,* representing the progression of the variable to the next state in the absence of any other operator that affects it.

The full P-SPO set, including environment operators, and updating utilities for multiple coins is:

$$\begin{aligned}
&flip(X):\{\} \rightarrow \left\{\begin{matrix} 0.5:showing(X,heads) \\ 0.5:showing(X,tails) \end{matrix}\right\}\ U(0.66) \\
&doNothing:\{\} \rightarrow \{\}\ U(0.66) \\
&environment:showing(X,heads) \rightarrow \{1.0:showing(X,heads)\}\ U(0.90) \\
&environment:showing(X,tails) \rightarrow \{1.0:showing(X,tails)\}\ U(0.47)
\end{aligned} \tag{9.2}$$

The utilities for the operators are calculated by averaging over the states to which they are applicable (see below). Notice that the *doNothing* action has no output, because the *environment* operators handle the evolution of perceptual features that are not affected by an action.

The utility attached to each of the other operators has changed to reflect the revised environment:

- *Both coins show heads:* the best action is to do nothing and receive the immediate reward (1.0).
- *One coin shows heads:* the best action is to flip the *tails* coin, with the result that the environment will be in the reward state with probability 0.5, or remain in the same state with probability 0.5. If the agent is not in a terminating state it can keep selecting the *flip* action until a reward is received. If there is a 0.9 discount for future rewards this results in a 0.91 value (see below).
- *Both coins show tails:* the best action is to flip either coin, with a probability of 0.5 of getting to the state in which one coin is showing heads (which has a utility of 0.91) and a 0.5 chance of showing tails, in which case we have a terminating state of two tails and a reward of 0.0. With discounts included, this is equal to: $0 \times 0.5 + 0.5 * 0.9 * 0.91 = 0.41$.

The values of each state can be calculated using Bellman updates:

$$V_{k+1}(s) = \max_a \sum_{s' \in S} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$$

$P^a_{ss'}$ = probability of moving from state s to s' given action a.

$R^a_{ss'}$ = reward received when action a is taken in state s and leads to state s'.

Terminating states are evaluated easily as: 0.0 for *both coins showing tails*; and 1.0 for *both coins showing heads*. For the *one coin showing heads* state, the maximum valued action is to *flip* the *tails* coin. The Bellman updates reach a stable state when the value of the state is = 0.91:

$$V_{k+1}(h,t) = \begin{cases} 0.5\times[1.0] \\ 0.5\times[0+0.9\times0.91] \end{cases} \tag{9.3}$$

The value of the two tails state is calculated by taking the optimal action of flipping one of the coins, which is the 0.5×0 + 0.5×0.82 (with 0.82 being the discounted value of the heads tails state above).

$$V_{k+1}(t,t) = \begin{cases} 0.5\times[0.0] \\ 0.5\times[0+0.9\times0.91] \end{cases} \tag{9.4}$$

The values of the non-terminating *doNothing* state-action pairs are equivalent to the best action from those states, multiplied by the discounts.

The output of the RVRL algorithm is an estimate of the aggregate of utilities of the state-action pairs which the P-SPOs match. The true aggregation can be taken by creating a state-action value map and finding the average of the state-action-pairs that match the operator's conditions.

The full state-action map, generated by performing Bellman updates in the state-action space, for the two coins environment is:

$$\begin{aligned}
&\textbf{\textit{doNothing, showing(penny, heads), showing(pound, heads) : U(1.0)}} \\
&\textbf{\textit{doNothing, showing(penny, heads), showing(pound, tails) : U(0.82)}} \\
&\textbf{\textit{doNothing, showing(penny, tails), showing(pound, heads) : U(0.82)}} \\
&\textit{doNothing, showing(penny,tails), showing(pound,tails) : U(0.0)} \\
&\textbf{\textit{flip(penny), showing(penny, heads), showing(pound, heads) : U(0.91)}} \\
&\textit{flip(penny), showing(penny,tails), showing(pound,tails) : U(0.41)} \\
&\textbf{\textit{flip(penny), showing(penny, tails), showing(pound, heads) : U(0.91)}} \\
&\textit{flip(penny), showing(penny,tails), showing(pound,tails) : U(0.41)} \\
&\textbf{\textit{flip(pound), showing(penny, heads), showing(pound, heads) : U(0.91)}} \\
&\textbf{\textit{flip(pound), showing(penny, heads), showing(pound, heads) : U(0.91)}} \\
&\textit{flip(pound), showing(penny,tails), showing(pound,heads) : U(0.41)} \\
&\textit{flip(pound), showing(penny,tails), showing(pound,tails) : U(0.41)}
\end{aligned} \tag{9.5}$$

For each operator, a utility can be found by finding the average of the values of the state-action pairs which match the operator conditions and action.

- The *flip* action has no conditions and therefore matches any of the sate-action pairs with flip. Its utility is: (0.91×4 + 0.41×4)/8 = 0.66.
- The *doNothing* action has no conditions, and matches any of the state-action pairs with the *doNothing* action. Its utility is therefore: (1.0+0.82×2+0.0)/4 = 0.66.

- The *environment* operator with condition *showing(X, heads)* matches all *heads* states for *doNothing*, and all *heads* states for which the flip *action* has been taken and the other coin was *heads.* These state-action pairs are shown in bold above. The utility is therefore = (1.0+0.82+0.82+0.91+0.91+0.91+0.91)/7 = 0.90.
- Similarly, the *environment* operator with conditions *showingTails* = (0.82 + 0.82 + 0.0 + 0.41 + 0.41 + 0.41 + 0.41)/7 = 0.47

RVRL uses an average of operator values to find the utility of taking an action. If, for example, the coins are in an initial state given by *{showing(tails, pound), showing(heads, penny)}*, then:

- The utility of taking the action of flipping the *pound* coin is the sum of the flip action and the environment operator with *heads* conditions = (0.66 + 0.90) /2 = 0.78.
- The utility of taking the action of flipping the penny coin is the sum of the flip action and the environment operator with *tails* conditions = (0.66 + 0.47)/2 = 0.57.
- The utility of the *doNothing* action is the sum of the two environment operators and the doNothing action = (0.9 + 0.47 + 0.66)/3 = 0.68.

The optimal action is, therefore, to *flip* the coin that is showing tails (with value 0.78).

9.2 Average Rule Value Update Function

The discussion of state aggregation above assumed that direct access to state values is available. If this was, indeed, the case then there would be no need to perform state aggregation because the full value-map could be used, rather than an estimate. RVRL is a method of estimating the state aggregation values based on further state aggregations.

Dynamic programming uses the Bellman update equation to continuously refine estimates of the value of being in a particular state, until equilibrium is reached. If the post decision state variable is used, then updates can equivalently be performed on state-action pairs.

The Rule Value Reinforcement Learning (RVRL) algorithm uses an approximate value iteration method to update a value associated with each rule, rather than each state. The main advantages of using a state-based aggregation method, such as RVRL, over a standard reinforcement learning technique are that:

- The agent does not have to store a value for every possible state-action combination in the environment.

- The agent can generalise over many states, with the result that one value can represent several states with similar properties, and a sensible action can be taken in previously unseen states.

The availability of the model (provided by the P-SPOs) means that full backups can be used, but we attach a learning rate so that the aggregation estimates are not unfairly biased towards the most recent update. Attaching a learning rate to Bellman updates gives an update function:

$$V_{k+1}(s,a) = V_k(s,a) + \sum_{s' \in S'} P^a_{ss'} \alpha [R^a_{ss'} + \gamma \max_{a'} V_k(s',a') - V_k(s,a)]$$

$$P^a_{ss'} = \text{probability of moving from state } s \text{ to } s' \text{ given action } a. \quad (9.6)$$

$$R^a_{ss'} = \text{reward received when action } a \text{ is taken in state } s \text{ and leads to state } s'.$$

The rule values for stochastic planning operators cannot be updated directly using the above equation because more than one rule will match the next state (*s'*) and maximum action (*a'*). The original RVRL algorithm (as presented in [16]) takes the *average* of the values attached to the P-SPOs that are used to generate the following states.

The rule learning function replaces *V(s',a')* with an average value for all matching rules which have precedence (and would therefore be used in generation of the successor state). The rules with precedence are used because they give the most accurate representation of the dynamics of the environment for that state and action.

The RVRL update function is given below. All rules matching the current state and action are updated with the new estimate of the following state.

```
RVRLUpdate(PSPOs, p, a)
for all (r ∈ matchingRules(PSPOs,p,a))
RV(r) = RV(r)+
   ∑ P^a_pp'α[R^a_pp' + γ max_a' avgRV(winningMatching(PSPOs,p',a')) − RV(r)]
  p'∈P'
```

Algorithm 9-1: RVRLUpdate(PSPOs, p, a). p = percept, a = action.

matchingRules(PSPOs, p, a) returns the rules with conditions matching the current state and action. *avgRV(winningMatching(PSPOs, p', a'))* returns the average value of the winning matching rules.

The two coin flipping example can be used to demonstrate this technique. The conditions captured in the rule-set for calculation of next state reflect structural characteristics of the environment for calculation of a value-map. The rule values are updated using Bellman updates using the rules matching the successor states. The *RV(r)* approximations are

initialised to a greedy estimate of 1.0 (the maximum achievable utility in an episodic environment with maximum reward of 1.0). The initial P-SPO set is therefore initially:

$$flip(X):\{\} \rightarrow \begin{Bmatrix} 0.5:showing(X,heads) \\ 0.5:showing(X,tails) \end{Bmatrix} U(1.0)$$
$$doNothing:\{\} \rightarrow \{\}\ U(1.0)$$
$$environment:showing(X,heads) \rightarrow \{1.0:showing(X,heads)\}\ U(1.0)$$
$$environment:showing(X,tails) \rightarrow \{1.0:showing(X,tails)\}\ U(1.0)$$

A single update takes a random initial state and a random action, uses the model to calculate the successor states and probabilities, and then updates the immediate rewards and the rule value estimates using the rules matching the successor states.

Taking an initial state of *{showing(penny, tails), showing(pound, tails)}* and an action *flip(penny)*, the possible successor states (evaluated using the P-SPOs as a model) are:

{showing(penny, tails), showing(pound, tails)} Pr(0.5)

{showing(penny, heads), showing(pound, tails)} Pr(0.5)

RVRL takes the value of the successor state to be the average of the rules that would be used if the maximum valued action was taken in that state (equivalent to dynamic programming). For the first state above, the optimal action is to flip either coin, which has the sum, equivalent to the update in (9.4), of:

$$V(\{showing(penny,tails),\ showing(pound,\ tails),flip(penny)\})$$
$$= Sum\begin{Bmatrix} 0.5\times[0.0] \\ 0.5\times[0+0.9\times Avg\begin{Bmatrix} flip(penny):1.0 \\ environment:showing(pound,tails):1.0 \end{Bmatrix} \end{Bmatrix}$$

The optimal action is again taken in the non-terminating successor state, giving an updated value estimate = 0.5×0+0.5×0.9×1.0=0.45.

If an updated rate, α, of 1.0 is used, then all rules matching the conditions (shown in bold) are updated with the new value estimate giving:

$$\boldsymbol{flip(X):\{\} \rightarrow \begin{Bmatrix} 0.5:showing(X,heads) \\ 0.5:showing(X,tails) \end{Bmatrix} U(0.45)}$$
$$doNothing:\{\} \rightarrow \{\}U(1.0)$$
$$environment:showing(X,heads) \rightarrow \{1.0:showing(X,heads)\}U(1.0)$$
$$\boldsymbol{environment:showing(X,tails) \rightarrow \{1.0:showing(X,tails)\}U(0.45)}$$

Continual updates of this form will improve the estimates, but will not settle on a value for each rule, in the way that dynamic programming would for state-action values, because each

rule is an aggregation of multiple states. These are, in effect, estimates of the state, so an update rule (similar to Q-learning updates) can be used to continuously improve the estimates.

The coin-flipping environment provides a simple example of the RVRL update function, but does not contain enough P-SPOs to demonstrate the issues involved in RVRL from a state-aggregation standpoint.

9.3 Rule Value Iteration

Chapter 5 described the process of building successor states using stochastic planning operators as a model. If this process is combined with the *RVRLUpdate* function, taking the current estimate of best action for each successor percept (section 9.4), it is possible to continuously generate next states from an initial state, and update the P-SPO values for operators matching those states until satisfactory values for the rules have been generated (or a fixed number of updates, *n*, have been performed). This process is described by the following algorithm:

```
ruleValueIteration(PSPOs,p,a)
initialise RV(r) = 0 for all r ∈ PSPOs;
repeat for n steps {
  {p'1,r1,pr1,...,p'n,rn,prn}←distributionModel(p,a)};
  totalValue = 0; totalReward = 0;
  for (i from 1 to n) {
    totalReward += ri×pri;
    a' = bestAction(PSPOs,p'i);
    maxActionValue = avgRV(winningMatching(PSPOs,p'i,a'));
    totalValue += maxActionValue × pr'i;
  }

  for each (r ∈ matchingRules(p,a))
     RV(r)=RV(r)+ α×(totalReward+γtotalValue − RV(r));
  p = pick a random percept from {p'1,...,p'n};
  a = random action;
}
```

Algorithm 9-2: ruleValueIteration. PSPOs=the planning operator set, p=initial percept, a=initial action. The algorithm takes a set of P-SPOs and iteratively improves the utility estimate associated with each operator for n-steps.

The sampling (temporal difference) equivalent of this method would take a sample next percept, *p'*, rather than calculating the probability of each successor percept. The process is otherwise the same.

A high value should be used for α initially, in order to remove initialisation bias from the rule value estimates. As the updates progress, a smaller α value will allow the rules to settle on the most accurate value for their state-aggregation. This can be achieved using the McLain formula for α (discussed in section 3.2.4). For the purposes of McLain updates, each P-SPO stores a count, *n*, of the number of times it has been updated. This ensures that rarely matched P-SPOs (those with specific conditions) are updated fairly in comparison to those with fewer conditions.

9.4 Best Action

The *bestAction* function tests the available actions for the given percept and returns the action with the highest average operator value. The operator values can be used as a policy by selecting the maximum valued action given an input percept.

```
bestAction(PSPOs,p)
   maxActionValue = -∞;
   maxAction = null;
   For (each a ∈ actions) {
      actionValue = avgRV(winningMatching(PSPOs, p,a));
      if (actionValue > maxActionValue)
      {
         maxActionValue = actionValue;
         maxAction = a;
      }
return maxAction;
```

Algorithm 9-3: bestAction. PSPOs=the planning operator set, p=initial percept. The algorithm returns the best action for the given percept.

If the environment is fully observable, then the percept, *p*, above can be substituted for a state, *s*.

9.5 Variance-Based Rule Value Evaluation Function

The use of an average across the winning matching P-SPOs to determine the approximate state-action values can be an effective method for environments in which each operator can be seen to have an equal contribution to the final state (as demonstrated in [16]). There are a range of situations, however, in which the aggregation given by the algorithm is not an effective estimate of the state-value or state-action value.

The main issues with using an average over P-SPO values are:

- *Accuracy-based weighting*: state-action utility estimates are not weighted towards the P-SPO which gives the most accurate prediction of expected future reward.
- *Bias:* P-SPOs utility estimate bias is not included in the weighting, with the result that operators with poor initial utility estimates have equal influence on the utility averaged across operators.

Both of these issues can be resolved by using a weighted average, favouring operators with the least variance in their estimates.

A further point is that the *samples* used to generate operator values need to be unbiased for the update rule to function correctly. Unbiased samples can be achieved by randomly selecting the state-action combination to be updated from the available state action space.

Aggregation based approximate dynamic programming methods store a set of value estimates that cover multiple states, or state-action pairs rather than a full look up table storing a value for each state-action combination. A key feature is that the state transition model can use the full set of features while the value function can be based on the aggregated state-action space. The model, in this research, is provided by the P-SPOs, which give a compact representation of the state-transition function. An aggregation function can, in the general case, be any sub-set of the features of state-action space. If the chosen sub-sets overlap this is referred to as a *soft aggregation* [73]. The conditions and actions in the P-SPOs are overlapping sub-sets of the features of the state-action space and we can therefore treat them as feature set selections for soft aggregation.

Equation (9.7) represents the equal weighting (average) estimate of value. The bar above the "*v*" in the equation is used to denote an estimate. The value is calculated as the mean of the utilities contained in all parallel stochastic planning operators used to generate the next state. The *RV(r)* function gives the utility estimate associated with the P-SPO, while *winningMatching(PSPOs, s, a)* returns the set of P-SPOs matching the state and action (after variables are resolved and conflicts removed).

$$\bar{v}_{(s,a)} = \sum_{r \in winningMatching(PSPOs,s,a)} \frac{RV(r)}{\left| winningMatching(PSPOs,s,a) \right|} \tag{9.7}$$

This was represented in the RVRL equation as:

$$\bar{v}_{(s,a)} = avgRV\left(winningMatching\left(PSPOs, s', a'\right)\right) \tag{9.8}$$

A possible weighting is to provide a weight based on the number of times the aggregation has been visited. This method can be used to overcome bias caused by states chosen initially for

update, but the aggregate measure becomes dependent on the distribution of states visited. Ideally, the weighting should reflect the accuracy of the operator in predicting the true value of the state-action pair, but, as this value is unknown, the accuracy compared to the current estimate is used. This can be achieved by finding the (estimated) variance of the error measure [73].

In general, a weighting of operators should sum to 1 and can be given as:

$$\overline{v}_{(s,a)} = \sum_{r \in WinningMatching(PSPOs,s,a)} w_r RV(r) \tag{9.9}$$

The equation tells us that there is an individual weighting, w_r, for each operator. The weighting can be dependent on some property of the operator. Equation (9.9) is equivalent to (9.7) if w_r is equal to $1/\left|winningMatching(PSPOs,s,a)\right|$.

The effectiveness of variance as a measure of accuracy can be demonstrated by considering an operator with no conditions and no action (e.g. a pure environment operator representing a random variable):

$$\{\}:\{\} \rightarrow \left\{ \begin{array}{l} .4:weather(sunny) \\ .3:weather(cloudy) \\ .3:weather(raining) \end{array} \right\}$$

This can be thought of as a maximum aggregation, because every state-action pair in the environment is contained within its aggregation set. When updating the utility estimate of this operator, we will, in some instances, gain exemplars of high valued state action pairs and at others low valued state-action pairs. The variance of the operator's estimate of the value, with respect to the observed values, will, therefore, be high. If, on the other hand, we have an operator which contains every state variable required to define the true value of an action, the estimated value will exactly match the observed values (if we have an accurate model and are using full, rather than sample, backups). The variance between the observed value and estimate will be zero.

In practice, we do not know in advance which of our aggregations reflect the true value, and the value estimates we are trying to improve are, themselves, based on aggregates of estimates. If the operators contain the required information for an effective aggregation, however, the variance will settle over time, with specific operators gaining weight (and influence) over general ones.

The weight aggregations contained in the P-SPOs matching the state and action, are each providing different estimates of the same quantity, $\overline{V}_{(s,a)}$. As stated by Powell in [73], statistics theory tells us that the weights that minimize the variance of $\overline{V}_{(s,a)}$ in equation (9.9) are proportional to the variance of the estimate, given by:

$$w_r \propto \left(\left(\bar{\sigma}_r^2 \right)^{(n)} \right)^{-1} \tag{9.10}$$

The weights need to sum to 1.0. We can therefore find a set of proportional weights for a set of operators by using:

$$w_r = \frac{\left(\sum_{r \in winningMatching(PSPOs,s,a)} \frac{1}{\left(\bar{\sigma}_r^2 \right)^{(n)}} \right)^{-1}}{\left(\bar{\sigma}_r^2 \right)^{(n)}} \tag{9.11}$$

The bracketed *n* above indicates that this is the weight for the *n*'th iteration of the update, rather than indicating an exponential.

An alternative weighting can be achieved by adjusting to remove the bias caused by the initial operator values. We can do this by using the total variation (the variance plus the square of the bias), giving the weights:

$$w_r = \frac{\left(\sum_{r \in winningMatching(PSPOs,s,a)} \frac{1}{\left(\bar{\sigma}_r^2 \right)^{(n)} + \left(\bar{\beta}_r^{(n)} \right)^2} \right)^{-1}}{\left(\bar{\sigma}_r^2 \right)^{(n)} + \left(\bar{\beta}_r^{(n)} \right)^2} \tag{9.12}$$

Equation (9.11) is the weighting used in *varianceRV,* the form used for the "variance update" tests in the results section, and (9.12) is *varianceBiasRV*, the form used in the "variance and bias update" tests in the results section (chapter 10). The use of bias relies on a more accurate aggregation being available as a reference. The soft aggregations in this research are not guaranteed to be more accurate and it is therefore interesting to evaluate which of these gives a more accurate value estimate. The initial tests indicate that there is no advantage in using the "variance and bias update" form of the equation, but further research is required in this area.

Both equations can result in zero values for variance initially and, therefore, division by zero errors. The equation used was amended slightly to give a minimum variance of 0.001 (a variance of less than 0.001 was substituted for 0.001 if it occurred in the evaluation).

9.6 Bias and Variance

A complete value map for a state-action space stores a current estimate for every state-action pair in the environment. The update equation gives a method for improving this estimate based on samples (evidence). The improved estimates will tend to increase in value each time if the value map has low (pessimistic) initial vale estimates, or tend to decrease in value if the initial estimates are high (optimistic). The positive or negative difference between the estimate and the real value is known as the *bias*. The variance is the mean squared error between the estimated and actual values.

The actual value for each state-action pair is not known, but the estimates of the value will improve over time. We can therefore estimate bias or variance by using an *update function* (9.13) on a current estimate of the bias or variance based on the new evidence. The aggregation estimates of utility contained in the rules also use an update function (iterations provide improving utility estimates).

Using a notation based on that used in [73], with a bar over a variable meaning it is estimated from sample observations, and a hat meaning a single observation, in general, an update function for an estimate, $\bar{\theta}$, of a value θ has the form:

$$\bar{\theta}^{(n)} = \left(1-\alpha^{(n-1)}\right)\bar{\theta}^{(n-1)} + \alpha^{(n-1)}\hat{\theta}^{(n)} \tag{9.13}$$

Where $\hat{\theta}^{(n)}$ is an unbiased observation of θ and is assumed to be independent of the previous estimate, $\bar{\theta}^{(n-1)}$. The learning rate α has the superscript (*n-1*) to allow for variable step-sizes (e.g. using McLain's updates). Both the variance and bias can be defined in terms of the error, ε, between the observation and actual value of θ.

$$\hat{\theta}^{(n)} = \theta^{(n)} + \varepsilon^{(n)} \tag{9.14}$$

If this error is treated as an exogenous measurement error between the actual value of θ and the observed value, the variance of $\bar{\theta}^{(n)}$ can be computed using equation (9.15).

$$\begin{aligned} &Var\left[\bar{\theta}^{(n)}\right] = \lambda^{(n)}\sigma^2 \\ &where\ \sigma^2 = Var[\varepsilon^{(n)}] \end{aligned} \tag{9.15}$$

$\lambda^{(n)}$ is a function of the step size (learning rate) in the update rule and can be computed using the recursion:

$$\lambda^{(n)} = \begin{cases} \left(\alpha_{(n-1)}\right)^2, & n = 1, \\ \left(1-\alpha_{(n-1)}\right)^2 \lambda^{(n-1)} + \left(\alpha_{(n-1)}\right)^2, & n > 1. \end{cases} \tag{9.16}$$

The value of the variance of the error, σ^2, is unknown, but it can be estimated from the data with each iteration of the update rule. The first step is to obtain an estimate of the bias, which can be found by using equation (9.17). This is, again, an update rule, giving an estimate of the (positive or negative) average difference between the estimate and actual value of θ. The update rate has been given as α for consistency of notation between update rules, but it must be noted that this is a separate step size and the subscript *v* is used to indicate that this value of α is shared between the bias and variance. The step size can be variable, but to simplify calculations, a fixed step size of 0.1 has been used.

$$\bar{\beta}^{(n)} = (1-\alpha_v)\bar{\beta}^{(n-1)} + \alpha_v(\bar{\theta}^{(n-1)} - \hat{\theta}^{(n)}) \tag{9.17}$$

To estimate the variance, we can use a similar update rule. The first step is to compute the total variance (including the bias), which can be estimated using a further update rule:

$$\bar{v}^{(n)} = (1-\alpha_v)\bar{v}^{(n-1)} + \alpha_v(\bar{\theta}^{(n-1)} - \hat{\theta}^{(n)})^2 \tag{9.18}$$

The variance of $\varepsilon^{(n)}$, σ^2, can now be calculated using the estimate of the total variance, by removing the influence of the bias:

$$\left(\bar{\sigma}^{(n)}\right)^2 = \frac{\bar{v}^{(n)} - \left(\bar{\beta}^{(n)}\right)^2}{1+\lambda^{(n-1)}} \tag{9.19}$$

The estimate of the variance of $\bar{\theta}^n$ can now be found by using $\left(\bar{\sigma}^n\right)^2$ in equation (9.15):

$$Var\left[\bar{\theta}^{(n)}\right] = \lambda^{(n)}\left(\frac{\bar{v}^{(n)} - \left(\bar{\beta}^{(n)}\right)^2}{1+\lambda^{(n-1)}}\right) \tag{9.20}$$

9.7 Variance Rule Value Iteration

Variance, or variance and bias, based updates can be incorporated into rule value iteration by associating extra values with each P-SPO (recording the current estimate of bias and/or variance), which are updated with each iteration. The two values can also be assigned associated learning rates and update counts for use with McLain's formula.

The variance rule value iteration function is given below. This is almost identical to the previous *ruleValueIteration* function, but:

(i) Replaces the *avgRV* function with *varianceRV* to find the state-action value estimate from the winning matching P-SPOs (using equation (9.11)).

(ii) Adds an update for the variance estimates, using the difference between the current rule value estimate and the observed (estimated) value (using equation (9.18)):

$$\bar{v}(r)^{(n)} = (1-\alpha_v)\bar{v}(r)^{(n-1)} + \alpha_v(RV(r) - totalReward + \gamma totalValue)^2$$

The equivalent update function for variance and bias replaces equation (9.11) in (i) with (9.12), and replaces equation (9.18) in (ii) with equation (9.20).

```
varianceRuleValueIteration(PSPOs,p,a)
initialise RV(r) = 0 for all r ∈ PSPOs;
repeat for n steps {
  {p'1,r1,pr1,...,p'n,rn,prn}←distributionModel(p,a)};
  totalValue = 0; totalReward = 0;
  For (i from 1 to n) {
    totalReward += ri×pri;
    a' = varianceBestAction(PSPOs,p'i);
    maxActionValue = varianceRV(winningMatching(PSPOs,p'i,a'));
    totalValue += maxActionValue × pr'i;
  }

  for each (r ∈ matchingRules(p,a)) {
     v̄(r)^(n) = (1−αv)v̄(r)^(n−1) + αv(RV(r) − totalReward + γtotalValue)^2;
     RV(r) = RV(r) + α^(r) × (totalReward + γtotalValue − RV(r));
  }
  p = pick a random percept from {p'1,...,p'n};
  a = random action;
}
```

Algorithm 9-4: variance RuleValueIteration. PSPOs=the planning operator set, p=initial percept, a=initial action. The algorithm takes a set of P-SPOs and iteratively improves the utility estimate associated with each operator for n-steps.

9.8 Optimistic Value Initialisation

Operators are initialised with *optimistic* value estimates to avoid local minima in the value space. An *optimistic* initialisation is one where all values are given an initial value of $R_{MAX}/(1-\gamma)$, which is the maxim value any state (or state-action pair) can reach under any policy. R_{MAX} is the maximum reward available in the environment. The maximum value for a state is equivalent to being in a state the gives the maximum reward, and continually taking an action that remains in that state. Feeding this in to the Bellman update equation it can be seen that the maximum value is dependent on the maximum immediate reward and the discount level for future rewards.

9.9 Summary

This chapter defined Rule Value Reinforcement Learning, an algorithm for associating state-action utility value estimates with P-SPOs. The basic form of the algorithm uses an average of the values associated with all *winning* operators applicable to a given state and action (winning operators are those that would be used to generate successor states). The variance based forms use the observation that operators with low variance give a consistent estimate and are, therefore, likely to be more accurate than those with a high variance. Weighting the total operator value in inverse proportion to the variance allows the most consistent estimators to have greater influence on the state-action value estimate. This observation is tempered by the fact that we do not have a real value of state value to base the estimates on.

Once a value has been associated with each P-SPO, a policy is implicitly contained within the estimates and can be extracted using the *bestAction* function, which iterates through all available actions finding the highest valued action according to the estimate contained in the *winningMatching* P-SPOs for the given percept.

10. Performance Results: Agent Framework with ASDD and RVRL

This section compares the results of the each of the update functions for the Rule Value Reinforcement Learning (RVRL) algorithm (defined in chapter 9). The updates are: aggregations by average; aggregation weighted by total variance; and aggregation weighted by variance (total variance with estimated bias removed).

Results are presented for the "slippery gripper" and "predator prey" environments. The P-SPO sets for all comparisons were learned using the ASDD algorithm (with set-based counting method). These operator sets were shown to produce accurate models and policy performances in comparison to MSDD in chapter 8. New perceptual data item sets (PDIs) were used to create the P-SPOs, and the performance of a tabular value map based policy for the new data set using the operators is given as comparison.

10.1 Slippery Gripper Environment

The following sections give the reward gathered by each RVRL update function, goals achieved vs. disaster states encountered, and examples of P-SPO sets with associated weights. The results in this environment are discussed in relation to the performance of a tabular value based policy.

10.1.1 Reward Gathered by a Policy Learned from the Data

Experimental conditions are unchanged from those used in section 8.2, with rewards set at +1 for a painted block delivered, and -10 for an unpainted block delivered. Policies were learned using dynamic programming for the tabular ASDD methods, or RVRL for all other methods, set to 10,000 iterations.

- Discount level, γ, is set to 0.9.
- No learning rate is required for the tabular method (equivalent to learning rate 1.0) because this is a model based technique with full backups.
- Learning rate for RVRL was set using McLain's formula for all methods, with a minimum value of 0.1: aggregations by average; aggregation weighted by total variance; and aggregation weighted by variance (total variance with bias removed).
- The policy for RVRL methods was taken to be the action found using the *bestAction* function (section 9.4).

Table 10-1 shows the reward gathered by an agent following a policy derived from each of the RVRL value update functions, compared with the ASDD tabular policy (which was found to be optimal with only limited experience). The table shows that RVRL, with either of the variance based value update functions, is able to learn an effective policy at all data levels. An optimal policy is learned at the 50,000 experience level. The average value function is ineffectual at learning a policy. The rule weights acquired appear to contain valid distinctions between good and bad actions, but overall, no useful action is taken.

Examining the action traces from these policies, the two effective policies can be summarised as:

- Optimal:
    - If the block is painted then use *new* (deliver it).
    - If the block is held then use *paint.*
    - If the block is not held and the gripper is dry, then use *pickup.*
    - If the block is not held and the gripper is not dry then use *dryer.*
- Non-optimal:
    - If the block is painted then use *new* (deliver it).
    - If the block is held then use *paint.*
    - If the block is not held then use *pickup.*

The non-optimal policy is identical except that the *dryer* action is not used. The reasons for this change can be attributed to a lack of information in the *dryer* action and are explored in section 10.1.3.

Table 10-1: Reward gathered after following a policy derived from a model learned from data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

	100	1000	5000	10000	20000	50000	100000
RVRL avg	0	0	0	0	0	0	0
RVRL var	14356	23304	14380	14390	14382	26096	26090
ASDD tab	26121	26129	26078	26127	26072	26120	26123
RVRL v+b	14242	14243	14364	14414	14196	26074	26101

Figure 10.1 shows the graph of these data values. There is an anomalous result at the 1000 PDI experience level, where the agent learns an effective, but not quite optimal policy. This

policy is essentially identical to the non-optimal policy above, but includes a rule that uses the reward as an input:

- Near-optimal:
    - If the block is painted then use *new* (deliver it)
    - If the block is held then use *paint*
    - If the block is not held then use *pickup*
    - If the reward is positive and the gripper is wet then use *dryer*

This policy is equivalent to the optimal policy under most conditions, because the block is not held just after the *new* action. If, however, the *dryer* action fails, the agent will not attempt to dry it again, and the gripper remains wet, resulting in several failed *pickup* actions.

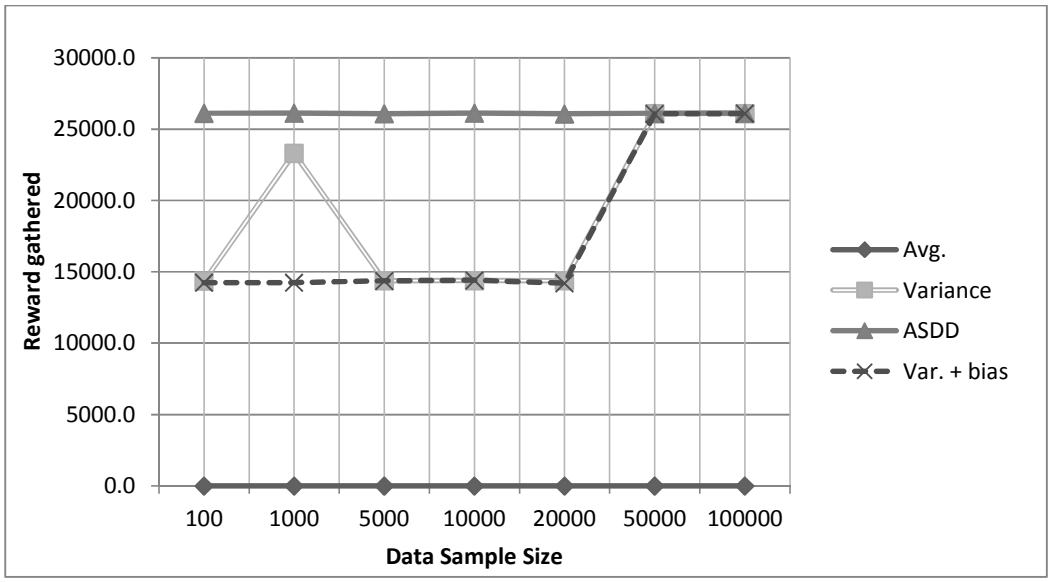


Figure 10.1: Graph of reward gathered after following a policy derived from a model learned from data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

10.1.2 Goals Achieved vs. Disaster States Encountered

There were no unpainted blocks delivered in any of the experiments using RVRL, and the reward gatherer for delivering a painted block is 1. Table 10-1 gives the goals achieved, with no need to represent disaster states encountered.

10.1.3 Examining Operator Weights for RVRL

The operator weights for the *average* aggregation demonstrate the problems with estimating rule values using this function. The biased reward, with -10 for a delivered unpainted block

and +1 for a delivered painted block, means that the average of the operators is affected strongly by the operator's negative values for using the *new* action in the situations in which the block is not painted. This explains the poor performance of RVRL with this update function in the domain.

Figure 10.2 shows the operator values (*U*) for the *new* operator with the average function, demonstrating that this operator would never be used under any conditions, because the negative values overpower the positives.

$$
\begin{array}{ll}
new: painted(false) & \rightarrow \{1.0: reward(neg)\} U(-9.98) \\
new: painted(true) & \rightarrow \{1.0: reward(pos)\} U(1.02) \\
new: \{\} & \rightarrow \{1.0: holding(false)\} U(-7.0) \\
new: \{\} & \rightarrow \{1.0: painted(false)\} U(-7.0) \\
new: \{\} & \rightarrow \left\{ \begin{array}{l} 0.3: dry(true) \\ 0.7: dry(false) \end{array} \right\} U(-7.0)
\end{array}
$$

Figure 10.2: New operator values after 10,000 iterations of RVRL with aggregation by average function for rules learned using ASDD from 100,000 PDIs in the slippery gripper environment.

The average for the above P-SPOs in situations for which the block is painted is (1.02-7.0-7.0-7.0)/4 = -4.995. The average for the situation in which the block is not painted is (-9.98-7.0-7.0-7.0)/4 = -7.75. Any rule with a higher value (such as using the *dryer*, or *pickup* action) is taken instead of the *new* action in all situations.

Figure 10.3 shows the operator values derived using *variance* update function, demonstrating that this method is able to distinguish situations in which the *new* operator is beneficial, and thus give a more accurate value estimate in these situations. There is more value in the rules in general, because an optimal policy has been learned by the system.

$$
\begin{array}{ll}
new: painted(false) & \rightarrow \{1.0: reward(neg)\} U(\text{-}7.69)\ \text{Var}(0.0003) \\
new: painted(true) & \rightarrow \{1.0: reward(pos)\} U(3.32)\text{Var}(0.0009) \\
new: \{\} & \rightarrow \{1.0: holding(false)\} U(\text{-}1.14)\text{Var}(23.94) \\
new: \{\} & \rightarrow \{1.0: painted(false)\} U(\text{-}1.14)\text{Var}(23.94) \\
new: \{\} & \rightarrow \left\{ \begin{array}{l} 0.3: dry(true) \\ 0.7: dry(false) \end{array} \right\} U(\text{-}1.14)\text{Var}(23.94)
\end{array}
$$

Figure 10.3: New operator values after 10,000 iterations of RVRL with aggregation by variance function for rules learned using ASDD from 100,000 PDIs in the slippery gripper environment.

The variance values above show that the algorithm tends toward the most accurate estimator of operator value. In this case, the operators with the *painted(true)* and *painted(false)* condition have the least variance and the alternative new operators are largely ignored in the operator value. The variance aggregation weight function (equation (9.10) from section 9.5) is used to combine the values:

$$w_{rule}^{(s,a)} = \frac{\left(\sum_{rule \in Winning(PSPOs,s,a)} \frac{1}{\left(\bar{\sigma}_{rule}^2\right)^{(n)}} \right)^{-1}}{\left(\bar{\sigma}_{rule}^2\right)^{(n)}}$$

The top of the equation is the inverse of the sum of the inverse of the variant estimate associated with each P-SPO. For a situation in which the *new* action is taken and the block is not painted, this gives:

= 1/(1/0.0003 +1/23.94 +1/23.94 +1/23.94)

= 1(3333.33 +0.04 +0.04 +0.04) =1/3333.45 = 0.0003

The weights for each component operator are therefore approximately:

1/3333.45 /0.0003 = 1.0 (for the operator with the *painted(false)* condition)

1/ 3333.45 /23.94 = 0.00 (for each of the other rules, which have no conditions)

The utility estimate for the *new* rule is taken to be approximately -7.69, that of the P-SPO:

$$new: painted(false) \rightarrow \{1.0: reward(neg)\} U(\text{-}7.69)\ \text{Var}(0.0003)$$

Similarly, the estimate for the *new* rule when the block is pained is approximately 3.32, that of the P-SPO:

$$new: painted(true) \rightarrow \{1.0: reward(pos)\} U(3.32) \text{Var}(0.0009)$$

As a comparison, the utility value generated by ASDD with tabular state-action values (approximately the actual value of the state) for any state in which the block is painted for the rule set is 3.02 (the value of taking the new action in this state). Tabular values for the *new* action when the block is not painted are not stored, because this is not the optimal action to take in the state. The value of the unpainted block states range from 2.16 to 2.72.

The next interesting point to consider is the difference in the rule values for those that learn an effective policy using the variance based methods and those that do not. The transition occurs

for P-SPO sets learned from 20,000 PDIs. Note that an effective policy can be gathered using ASDD with tabular utility values (using the operators as a model) much earlier than this. The change must, therefore be due to differences in the rules, or in the associated values, rather than the modelling ability.

The key P-SPOs are those for the *dryer* action and *pickup* action. An optimal policy must be able to recognise that the *dryer* action has greater value than *pickup* if the block it not painted and the block is not dry. The non-optimal policy will continue to attempt to pick up the block, even of the block is not dry. An example percept for this situation is:

$$painted(false), clean(true), dry(false), holding(false), reward(pos)$$

Figure 10.4 shows the P-SPO set for the *dryer* action for the above percept, for the operator set learned from 20,000 PDIs.

$dryer : \{\} \rightarrow \{1.0 : reward(none)\} U(2.76)$ Var(0.0816)

$dryer : painted(false) \rightarrow \{1.0 : painted(false)\} U(2.61)$Var(0.0130)

$dryer : clean(true) \rightarrow \{1.0 : clean(true)\} U(2.68)$Var(0.0426)

$dryer : dry(false) \rightarrow \left\{ \begin{matrix} 0.91 : dry(true) \\ 0.09 : dry(false) \end{matrix} \right\} U(2.78)$Var(0.0747)

$dryer : holding(false) \rightarrow \{1.0 : holding(false)\} U(2.75)$Var(0.0844)

Total Variance Weighted Action Value: 2.664

Figure 10.4: P-SPO set for the dryer action with variance weighted rule values for an operator set learned from 20,000 PDIs.

The difference in variance values for the above operators is not as pronounced as for the *new* operator shown earlier, reflecting the fact that the operators' conditions contain similar amounts of information. The overall rule value is biased towards the weight of the operator with the *painted(false)* condition, because this condition is the best estimator of the value of the state (versus those in which the block is painted). This highlights an issue with state aggregation of this type: the effectiveness of the dryer action depends on all of these factors equally, but the painted condition is the best indicator of the current state value given the limited information.

Figure 10.5 shows the P-SPO set for the *pickup* action for the operator set:

$$pickup:\{\} \rightarrow \{1.0:reward(none)\}U(2.82)\ \text{Var}(0.054)$$
$$pickup:painted(false) \rightarrow \{1.0:painted(false)\}U(2.74)\text{Var}(0.022)$$
$$pickup:clean(true), painted(false) \rightarrow \{1.0:clean(true)\}U(2.74)\text{Var}(0.022)$$
$$pickup:dry(false) \rightarrow \{1.0:dry(true)\}U(2.71)\text{Var}(0.069)$$
$$pickup:holding(false), dry(false) \rightarrow \begin{Bmatrix} 0.84:holding(false) \\ 0.16:holding(true) \end{Bmatrix} U(2.66)\text{Var}(0.06)$$
Total Variance Weighted Action Value: 2.73

Figure 10.5: P-SPO set for the pickup action with variance weighted rule values for an operator set learned from 20,000 PDIs.

Notice that the minimum variance operators for the action are, again, those with the *painted(false)* conditions. This makes intuitive sense, in that there is no reason to pick a block up if it is already painted, but shows the difficulty in learning operator values from limited information.

The difference between the *dryer* action value (2.664) and *pickup* action value (2.73) is minimal. The optimal action is *dryer* but the aggregation is not able to pick this up because there is a lack of information in the conditions with which to make a distinction.

The P-SPO set for the *dryer* action and *pickup* action for the situation where the block is not painted, not held, and not dry, for the rule set learned from 50,000 PDIs, are investigated for comparison.

Figure 10.4 shows the P-SPO set for the *dryer* action for the above percept, for the operator set learned from 20,000 PDIs.

$$dryer:\{\} \rightarrow \{1.0:reward(none)\}U(2.29)\ \text{Var}(0.0513)$$
$$dryer:painted(false) \rightarrow \{1.0:painted(false)\}U(2.58)\text{Var}(0.0247)$$
$$dryer:clean(true) \rightarrow \{1.0:clean(true)\}U(2.61)\text{Var}(0.0272)$$
$$dryer:dry(false) \rightarrow \begin{Bmatrix} 0.9:dry(true) \\ 0.1:dry(false) \end{Bmatrix} U(2.70)\text{Var}(0.0579)$$
$$dryer:holding(false) \rightarrow \{1.0:holding(false)\}U(2.59)\text{Var}(0.0448)$$
Total Variance Weighted Action Value: 2.623

Figure 10.6: P-SPO set for the dryer action with variance weighted rule values for an operator set learned from 20,000 PDIs.

Figure 10.5 shows the P-SPO set for the *pickup* action for the rule set:

$pickup:\{\} \rightarrow \{1.0:reward(none)\}U(2.75)$ Var(0.091)

$pickup:painted(false) \rightarrow \{1.0:painted(false)\}U(2.62)$Var(0.033)

$pickup:clean(true), painted(false) \rightarrow \{1.0:clean(true)\}U(2.63)$Var(0.034)

$pickup:dry(false) \rightarrow \{1.0:dry(true)\}U(2.58)$Var(0.046)

$$pickup:holding(false), dry(false) \rightarrow \begin{Bmatrix} 0.86:holding(false) \\ 0.14:holding(true) \end{Bmatrix} U(2.53)\text{Var}(0.042)$$

Total Variance Weighted Action Value: 2.61

Figure 10.7: P-SPO set for the pickup action with variance weighted rule values for an operator set learned from 20,000 PDIs.

The operator value distinction is again minimal, but in this instance the optimal action has a small advantage and would, therefore, be taken. Notice that the rule sets and conditions for the above P-SPOs taken from 50,000 trails are identical to the rules and conditions for the P-SPOs taken from 20,000 trails. The only difference being the probabilities for the outcomes. These probabilities show only small changes, leading to the conclusion that RVRL methods are unlikely to be reliable at learning effective conditions for rules with this minimal level of information in the conditions. One solution to this is to add conditions when the variance between rules does not drop below a threshold. This is an area for further investigation and is discussed in section 11.3.

The ASDD-tabular value for states matching these operators, in which: the block is not painted, not held and the gripper is not dry is 2.16. The state value estimates are over-estimates because the average value in the system is greater than 2.16, so rules with fewer conditions (such as those for the *pickup* and *dryer* actions) will aggregate over all rules they apply to, such as using the action when the block is already painted. Although not the optimal action, the state itself is of high value, as is the successor state.

10.1.4 Discussion of Slippery Gripper RVRL Results

The RVRL with *variance* method is able to learn an effective, although not reliably optimal, policy for the environment. The algorithm is able to overcome the issues that are present with the *average* method to identify the operator with the best estimate of sate value in a particular situation, but the operators lack the information required to perform completely accurate state aggregation. The results show, however, that an effective estimate of the utility of taking an action in a given state has been learned by the technique. This is an encouraging result and indicates that RVRL based techniques could be used as a basis for future algorithms which refine the operator conditions when a low variance has not been achieved.

10.2 Predator Prey Environment

The following sections present the rewards gathered by each RVRL update function, and examples of operators with associated weights, of the "predator-prey" environment. The results in this environment and discussed in relation to the performance of a tabular value based policy.

10.2.1 Reward Gathered by a Policy Learned from the Data

Experimental conditions are unchanged from those used in section 8.3, with rewards set at +1 for capturing the prey and 0 otherwise. Policies were learned using dynamic programming for the tabular ASDD methods, or RVRL for all other methods, set to 10,000 iterations.

Experimental conditions were:

- Discount level, γ, was set to 0.9.
- No learning rate is required for the tabular method (equivalent to learning rate 1.0) because this is a model based technique with full backups.
- Learning rate for RVRL was set using McLain's formula for all methods, with a minimum value of 0.1: aggregations by average; aggregation weighted by total variance; and aggregation weighted by variance (total variance with bias removed).
- The policy for RVRL methods was taken to be the action found using the *bestAction* function (section 9.4).

Table 10-2: Reward gathered after following a policy derived from a model learned from data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

	100	1000	5000	10000	20000	50000	100000
RVRL avg	6045.00	9230.00	6929.00	6397.00	7716.00	6006.00	6001.00
RVRL var	6394.00	6394.00	7141.00	6344.00	13570.00	13669.00	15834.00
ASDD tab	6518.00	11267.00	8798.00	10083.00	13605.00	14537.00	15838.00
RVRL var+bias	4852.00	7578.00	7226.00	5113.00	12957.00	12065.00	15369.00

Figure 10.8 shows the graph of these data values. The RVRL methods using *variance* and *variance with bias* based aggregation are effective at learning an equivalent policy to the tabular policy learned by ASDD with experience from 20,000 PDIs or greater.

RVRL with *average* based aggregation was not able to learn an effective policy in the majority of cases, although there is a peak in performance at 1000 PDIs data sample size. It should be noted that RVRL with average based aggregation has been shown to be effective at

learning a policy in this environment in previous work, but that the balance between reward and punishment in the environment in this case was set such that operators averaged to near zero values [15]. This appears to be a specific case, with variance based methods showing more generally promising application.

The ASDD tabular method represents a maximum that could be expected from RVRL, because this is an optimal policy given the imperfect model generated by P-SPOs with the assumption of independent outputs (discussed in section 8.3).

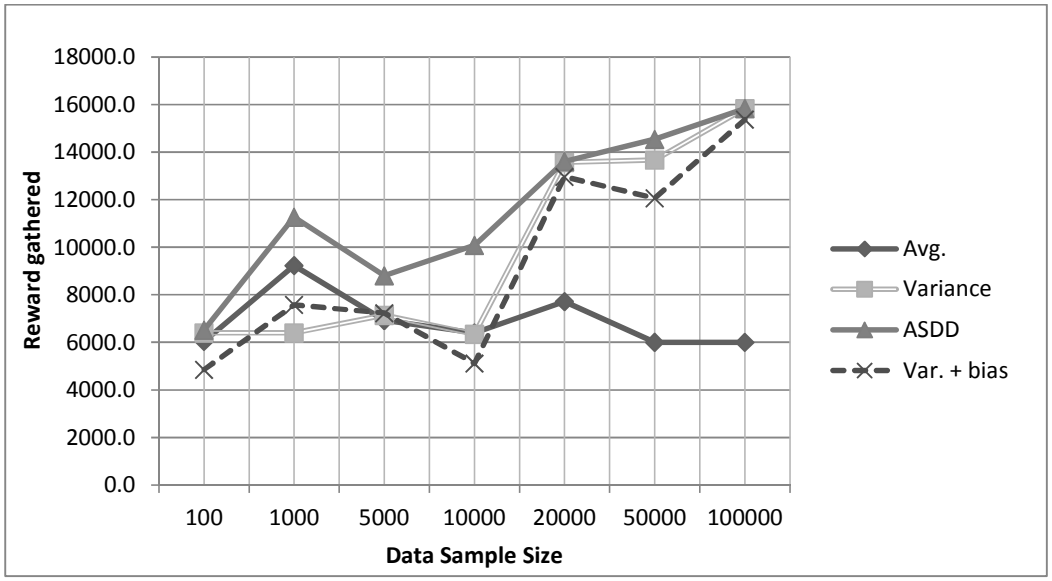


Figure 10.8: Graph of reward gathered after flowing a policy derived from a model learned from data collected from 100, 1000, 5000, 10000, 20000, 50000 and 100000 random moves.

A point of interest in the graph is change in the ability of variance based methods to learn an effective policy between 10,000 and 20,000 PDIs. The model is sufficient to learn an effective policy, as shown by the ASDD tabular results, but RVRL has not performed well. This is caused by a lack of information in the operator conditions. Examples of this are given in the following section.

10.2.2 Examining Operator Weights for RVRL

The key distinction in the learning ability of variance based aggregation methods is between the operators learned from 10,000 and 20,000 PDIs. Examination of these operators shows that the 20,000 PDI operators contain extra conditions, and therefore represent finer grained distinctions between states for the aggregation process. The accuracy of RVRL with variance base methods in predicting a state value is reflected in the variance levels of the operators.

An example of the winning operators and associated weights for the 20,000 and 10,000 PDI data sets is presented below, for an initial action *move(west)* and initial percept:

$$see(north, empty), see(east, empty), see(south, empty), see(west, wall), see(under, agent)$$

This is the situation in which the predator is on-top of the prey, has a wall to its west, and moves into the wall (resulting in it staying in the same position).

Figure 10.9 shows the P-SPOs acquired from the 20,000 PDI data set. The third operator has the lowest variance. It contains the conditions that the agent is in the *under* position, two adjacent squares are empty, and the predator is moving away from an empty square (into an unknown square which cannot contain the agent). This operator is the most specific, in that it refers to the least frequently occurring states in the model because the *agent(under)* condition is relatively rare, despite having fewer conditions than the 2nd operator in the set. The overall variance weighted value is dominated by this operator's value.

$$move(west) : see(north, empty) \rightarrow \left\{ \begin{array}{l} 0.05 : see(north, agent) \\ 0.95 : see(north, empty) \end{array} \right\} \begin{array}{l} U(1.08) \\ \text{Var}(0.0441) \end{array}$$

$$move(west) : \begin{array}{c} see(north, empty), see(east, empty) \\ see(south, empty), see(west, wall) \end{array} \rightarrow \left\{ \begin{array}{l} 0.09 : see(east, agent) \\ 0.91 : see(east, empty) \end{array} \right\} \begin{array}{l} U(1.03) \\ \text{Var}(0.0252) \end{array}$$

$$move(west) : \begin{array}{c} see(east, empty), see(south, empty) \\ see(under, agent) \end{array} \rightarrow \left\{ \begin{array}{l} 0.09 : see(south, agent) \\ 0.91 : see(south, empty) \end{array} \right\} \begin{array}{l} U(1.24) \\ \text{Var}(0.0009) \end{array}$$

$$move(west) : see(west, wall) \rightarrow \{1.0 : see(west, wall)\} U(1.11) \text{ Var}(0.0245)$$

$$move(west) : see(under, agent) \rightarrow \left\{ \begin{array}{l} 0.3 : see(under, agent) \\ 0.7 : see(under, empty) \end{array} \right\} \begin{array}{l} U(1.28) \\ \text{Var}(0.0102) \end{array}$$

$$\text{Overall Variance Weighted Value} : 1.23$$

Figure 10.9: P-SPOs acquired from 20,000 PDIs for the predator-prey environment.

The overall values for each move for the initial percept are:

move(north): 1.18.

move(east): 1.13.

move(south): 1.14.

move(west): 1.23 (shown in detail above).

If these P-SPOs are compared to those for the10,000 PDI data set then the lack of information in the 10,000 PDI set becomes clear. The rules have fewer conditions, and thus represent broader aggregations of states. The overall value of the P-SPOs is similar, but with less information in the rules. The predator is not, therefore, able to correctly evaluate differences between the values of taking each action in the state in comparison to other states.

$$move(west) : see(west, wall), see(under, agent) \rightarrow \left\{\begin{array}{l} 0.35 : see(north, agent) \\ 0.54 : see(north, empty) \\ 0.11 : see(north, wall) \end{array}\right\} \begin{array}{c} U(1.17) \\ \text{Var}(0.0005) \end{array}$$

$$move(west) : \begin{array}{c} see(east, empty), see(south, empty) \\ see(under, agent) \end{array} \rightarrow \left\{\begin{array}{l} 0.13 : see(east, agent) \\ 0.87 : see(east, empty) \end{array}\right\} \begin{array}{c} U(1.19) \\ \text{Var}(0.0002) \end{array}$$

$$move(west) : see(south, empty) \rightarrow \left\{\begin{array}{l} 0.05 : see(south, agent) \\ 0.95 : see(south, empty) \end{array}\right\} \begin{array}{c} U(1.01) \\ \text{Var}(0.0158) \end{array}$$

$$move(west) : see(west, wall) \rightarrow \{1.0 : see(west, wall)\} U(1.05)\ \text{Var}(0.0263)$$

$$move(west) : see(under, agent) \rightarrow \left\{\begin{array}{l} 0.28 : see(under, agent) \\ 0.72 : see(under, empty) \end{array}\right\} \begin{array}{c} U(1.19) \\ \text{Var}(0.0002) \end{array}$$

Overall Variance Weighted Value : 1.18

Figure 10.10: P-SPOs acquired from 20,000 PDIs for the predator-prey environment.

A point of interest in the above operators is that three of them have variance less than 0.001 (the minimum variance in the update equation) and will therefore give equal contributions to the output weight. This reflects the fact that the aggregations are too general and the system is settling towards an average weight of rules in which the predator is under the prey.

The overall values for each move for the initial percept are:

move(north): 1.16

move(east): 1.19

move(south): 1.15

move(west): 1.18

There is little difference between the values of these actions, which is reflected in the performance of the predator in being approximately equivalent to random moves.

10.2.3 Discussion of RVRL Predator Prey Results

The RVRL with *variance* method is able to learn an effect policy under certain conditions, but is not able to learn an optimal policy in the environment under any level of experience. This is due to inaccuracies in the model acquired by the rules rather than issues with RVRL itself. The tabular ASDD method finds the optimal policy possible from the P-SPO based model, and the variance-weighed RVRL models are able to match the performance of this model under certain conditions. The average weighted RVRL method was not able to acquire an effective policy in this environment; although it has been shown to be effective under specific conditions in previous work (positive and negative rewards must balance in the environment).

Examining the P-SPOs and associated values shows that the key to the ability of RVRL to acquire an effective policy with variance-based RVRL is the information contained in the

conditions of the operators. There is a step change between the rules built from 10,000 and 20,000 PDIs. The model provided by the rules at these levels is not markedly more accurate, but the additional information contained in the rules allows finer grained state aggregations to be made.

This is, again, an encouraging result and indicates that RVRL based techniques can be used to form effective policies by attaching utility values to P-SPOs. An interesting avenue for future work is to explore methods of adding conditions in situations in which the operators give inaccurate estimates. With no obvious method of knowing whether the estimate is inaccurate it is tempting to add conditions if the variance of a rule-set is high, but the experiments show that low variance estimates can still provide poor estimates of state-value. An alternative is to investigate the number or operators which display low variance, taking care to ignore multiple operators with the same conditions.

Variance and variance-with-bias methods show some differences in results, but further experimentation is required to evaluate the reasons for this. Early indications are that a lack of finer grained aggregation in the target values for the variance-with-bias method has the result that the method is no more effective than using variance alone (a simpler method to calculate).

10.3 Summary

This section investigated the performance of RVRL with three alternative operator weightings against a standard tabular dynamic programming method using the same P-SPO based model. The methods were evaluated in the “slippery gripper” and “predator-prey” environment. The results show that variance based RVRL methods are able to learn effective policies under a range of training experience levels, while average weighting is not effective in the general case.

Investigation of the weights and values associated with each P-SPO indicate that the performance of RVRL is dependent upon the information contained in the P-SPOs. A lack of correlation between operator conditions and utility structure in the environment will make RVRL ineffective, while highly specific operators provide fine-grained state aggregations which can be exploited to form an effective policy.

11. Conclusions

This thesis presented a framework for the creation of autonomous agents that are able to model a stochastic environment, plan within the model, and act on the plan. The scope of the work is ambitious and this section investigates the extent to which this ambition has been achieved, with comparison to related work and thoughts on future directions for the research.

The research presented the BatchModelQ framework for environment modelling agents, based on the Dyna-Q framework, incorporating either approximate or exact dynamic programming methods for policy formation in the environment.

The hypothesis, stated in chapter 1, is that utility estimates attached to acquired parallel stochastic planning operators, describing the dynamics of a predictably probabilistic environment, can be used to compactly model the effectiveness of taking actions in that environment.

This was to be accomplished by a set of sub-objectives, and the achievements with respect to of each of these will be evaluated.

11.1 Review of Objectives

The first objective was to "*create a framework for environment modelling agents*". The BatchModelQ framework, based on Dyna-Q, was described in chapter 4. The framework is adaptable in the sense that it is agnostic to the environment modelling system used, requiring only that the model is able to predict future percepts (or equivalently states, in fully observable environments), given an input percept and action. The policy learning stage in the model is also agnostic to the policy learning system used, and is able to incorporate dynamic programming, Q-learning, RVRL or any equivalent system. The action selection stage extracts the best action for the given percept by querying the policy formed. The main restrictions on the framework are that: the environment modelling is a batch process; and the policy is deterministic. Extensions to the system to remove these restrictions are investigated in section 11.3.

The second objective was to "*design a rule-based environment modelling system*". Parallel Stochastic Planning Operators (P-SPOs) were defined in chapter 5, including algorithms for percept/state generation. P-SPOs have the expressive power to compactly model a stochastic environment from the perspective of an agent. The operators model the environment's response to an agent action and its evolution outside the agent's control with the use of *environment* operators. The syntax includes the ability to apply multiple operators in parallel, which allows actions with independent outcomes to be modelled with a minimal set of

operators. The operators have an advantage over graphical model representations of being human readable and therefore open to interpretation and modification.

The third objective was to *"design a system for learning the rule-based environment modelling system from experience"*. Chapter 6 defined ASDD, a batch process for the automated acquisition of a P-SPO set from experiential data, based on the Apriori method of association rule mining, combined with a G statistic test to filter rules with conditions which do not have a significant influence on operator outcomes. Supplemental algorithms combine mined rules into P-SPOs and add missing rules for P-SPOs whose outcome probabilities do not sum to 1.0. The method does not, however, build P-SPOs with the full expressive power of the defined syntax, with acquisitions of variable substituted operators and operators with dependent outcomes being a subject for future work. The current batch learning process is also restrictive and extensions for in-line learning are an area for extension of the algorithm. Chapter 8 evaluated the performance of the ASDD algorithm, showing it to be capable of learning a perfect P-SPO based model of an environment with independent outcomes, and an effective model for an environment with dependency between outcomes. The algorithm was shown to be fast in comparison to MSDD and the model generated by the P-SPOs was shown to be effective as a basis for building a dynamic programming based policy. Parallel extensions of ASDD to further improve performance are discussed in section 11.3.2.

The fourth objective was to *"design a system for attaching utility estimates to the rules"*. The RVRL algorithm was defined in chapter 9. The system iteratively updates rule values associated with each P-SPO. The conditions of the P-SPOs are treated as state aggregation utility estimates and are combined using a weighted average of the P-PSOs which match a given state-action pair. The system builds rule utility estimates from successor rule utility estimates, without the need to enumerate the value of every state, or state-action pair in the environment. Chapter 10 presented experimental results, showing that a policy formed from the RVRL utility estimates contained in P-SPOs could be optimal, given sufficient environmental experience, in the "slippery gripper" domain. The RVRL based policy for the "predator-prey" environment was optimal, given sufficient information in the operator conditions, with respect to the modelling power of the operators themselves (without dependencies between outcomes).

Overall, P-SPOs were shown to be effective in modelling an environment for which the Markov property holds and conditional independence exists between the output variables. ASDD was shown to be an effective learning method for P-SPOs in environments of this type. As discussed in the future work sections below, the modelling capacity of P-SPOs is

likely to be sufficient to model all Markov environments with refinements to the ASDD algorithm to enable learning of operators which model dependencies between outcomes. Currently the system is able to learn an approximate model with the addition of background knowledge for an environment with multiple dependencies between outcomes.

The complete system has been shown to perform well in situations for which limited environment experience has been gathered and is able to outperform a tabular model under these conditions.

The use of acquired stochastic planning operators, combined with RVRL, represent a promising development in approximate dynamic programming. Results for the "slippery gripper" environment demonstrated the value of generalisation when the agent has incomplete knowledge, with the agent able to learn a policy with only limited environmental experience.

The experiments presented in this thesis represent an early indication as to the full effectiveness of the approach. It is anticipated that some classes of environment will show particular improvements in space requirements for both the model and the policy over explicit state-space environments. Take, for example, a simple extension of the single coin-flipping agent introduced in chapter 1 as follows.

If we extend the agent's percept to include a history of length n of the state of the coin after each previous flip, then the size of the agent's percept space increases exponentially with the history length to the of size 2^n. A tabular model of this environment would require an entry for each percept-action pair, with the two actions being *flip* and *doNothing,* giving 2×2^n entries. Each entry would require a successor percept, which, for *doNothing* is a single entry, and for *flip* would require two successor percepts with associated probabilities, giving a total of $2\times2^n\times2/2+2\times2^n\times1/2 = 2\times2^n\times1.5$ successor state entries.

The state-value map for the environment is of size 2^n, while the state-action value map is of size 2×2^n.

Contrast this with a P-SPO representation of the model. Using an equivalent representation, each additional entry in the history (h_n) requires two further operators:

$$\{\}: showing(coin_{h(n)}, heads) \rightarrow \{1.0: showing(coin_{h(n+1)}, heads)\}$$
$$\{\}: showing(coin_{h(n)}, tails) \rightarrow \{1.0: showing(coin_{h(n+1)}, tails)\}$$

For a history of length 1, the full operator set would be:

$$flip(coin):\{\} \rightarrow \left\{\begin{matrix} 0.5:showing(coin,heads) \\ 0.5:showing(coin,tails) \end{matrix}\right\}$$
$$doNothing:showing(coin,heads) \rightarrow \{1.0:showing(coin,heads)\}$$
$$doNothing:showing(coin,tails) \rightarrow \{1.0:showingTails(coin,tails)\}$$
$$\{\}:showing(coin,heads) \rightarrow \{1.0:showing(coin_{h1},heads)\}$$
$$\{\}:showing(coin,tails) \rightarrow \{1.0:showing(coin_{h1},tails)\}$$

The size of this operator set increases by 2 for each additional history length, and is therefore of linear size: $2\times n+3$, for a history of length n. RVRL associates a utility with each operator and requires a set number of entries for each operator, dependent upon the weighting algorithm used. Using a variance-based weighting, the majority of the operators will settle on an average utility and have high variance, because the operators have no predictive power for the outcome coin state. The *doNothing* and *flip* operators will settle on approximately the same utility as the history length zero equivalents.

This example shows that, for environments displaying conditional independence in successor state features, the approach presented in this thesis is linear in both model size and policy space requirements, while equivalent tabular methods are exponential in size.

Some limitations to the current approach have also been demonstrated. The key issue for the effectiveness RVRL is the *availability of information in the operator conditions*. If the P-SPO set models the environment compactly, then the state-space is split into a small number of large, overlapping, aggregations by the operator conditions. This means that the RVRL-based value approximations are not able to find perfect policies for some environments. This is a major limitation of the system, but was anticipated because the more compact an operator set is at modelling an environment, the less specific the information contained within the operators becomes. Approaches to augment the rules with additional information in such circumstances are investigated in section 11.3. Despite this limitation the system can form effective, if not reliably optimal, policies and the storage overhead required is minimal.

11.2 Comparison with Related Work

The three main contributions of this research are in the areas of P-SPO definition, P-SPO learning, and approximate dynamic programming using state-aggregation.

Related work in the field of planning operators is extensive, and the discussion in this section is therefore restricted to stochastic operators for which learning from data is possible, briefly discussing other types of operators and why they cannot be used.

Methods for learning the P-SPO set are related to stochastic logic program learning, planning operator learning, and two-tier Bayesian networks (2TBNs) learning (defined in influence map form in chapter 3).

Techniques for resolving the "curse of dimensionality", the need to store a number of state-action values exponential to the number of variables in the state, fall into two main categories:

- State-based aggregation.
- Functional Approximation.

RVRL is a state-based aggregation technique, in that states which behave in a similar way with respect to a given action, conditions, and goal are given the same value. This type of aggregation is captured within the P-SPO values by RVRL. Other techniques in this category include decision theoretic regression [10], explanation based reinforcement learning [24] and relational reinforcement learning [29].

Functional approximation techniques seek to create a compact approximation to the value function using, for example, neural networks. This technique gained prominence with TD-Gammon, which created a championship winning backgammon program [91]. The technique uses an approximation to the value function, rather than exploiting regions of uniform value in the feature space. Full comparisons with these techniques are not discussed, but are a subject for future research.

11.2.1 Planning Operators

This section brief discusses alternative probabilistic planning operator representations. The main criterion is that these should use a *possible worlds* and *discrete time* representation, because the state at each time slice can be used as training data for a learning algorithm. This restriction means the event calculus [47] and its probabilistic extensions are beyond the scope of this analysis. Recent advances in learning probabilistic event calculus rules have been made using the probabilistic logic network (PLN) framework [37].

The Independent Choice Logic

The Independent Choice Logic (ICL) is a planning logic system for modelling action under uncertainty [70], which provided initial inspiration for the P-SPO definition.

The ICL gained prominence with application to multi-agent systems when it was demonstrated that it could be used to model the actions of agents and the environment using a compact logic [71]. The semantics define a set of independent choices with a probability distribution for each, and a logic program which defines the possible outcomes if the choice is taken in terms of *possible worlds*. A possible world is defined, as in this thesis, as a single

configuration of the environment at a discrete time interval, with each element being in a defined state. The probability of each possible world is determined by the initial world, the independent choices made, and the logic program. Further possible worlds can be determined from each possible world, enabling planning.

The full definition of the IPL implements game theoretical aspects, which are beyond the scope of this work, and the embedding of influence diagrams and Markov decision processes in the logic. Rather than investigate these points, a short example (taken from [71]) will demonstrate the main issue in terms of learning an ICL rule-set.

A *coin toss* environment can be modelled in the ICL as:

$$
\begin{aligned}
&heads(C,T+1) \leftarrow \\
&\quad tossed(C,T) \wedge \\
&\quad heads\_turns\_up(C,T) \\
&heads(C,T+1) \leftarrow \\
&\quad \neg tossed(C,T) \wedge \\
&\quad heads(C,T) \\
&tails(C,T) \leftarrow \\
&\quad \neg heads(C,T)
\end{aligned}
$$

The agent's choice of action is represented by the *tossed* condition, and the random result of the action is represented by the *heads_turns_up* condition. *heads_turns_up(C,T)* is *true* at time *T+1* if coin *C* would show heads if it was tossed at time *T*. The probability of *heads_turns_up(C,T+1)* being *true* is 0.5.

While it is possible for a rule designer to define a rule set containing conditions which represent probabilities, these conditions are not, in general, observable in the environment and cannot, therefore, be modelled from data. In the example above, the agent could observe whether a *head* was present in a successor state, but without prior knowledge of the probabilistic conditions, it is not possible to infer them from data to make rules of this type.

The ICL is, however, a powerful formalism, able to model other agent's actions in the same way as the *heads_turns_up* condition, to indicate an event beyond the agent's control. Recent advances to the ICL have addressed issues with existence and identity in possible worlds [72], and lifted inference: inference that leaves variables un-instantiated for as long as possible using a state-aggregation based approach [45].

A key advantage that the ICL has over other stochastic planners is that it is able to model game theoretical actions, such as that of a football goal-keeper's decision to dive left or right

when attempting to save a penalty. This allows the formation of game-theoretic policies, which are currently not possible in a P-SPO-based approach. Further investigation of this could prove valuable in finding computer-game applications for P-SPOs, because predictable actions are often not sufficient for an opposition AI in a stochastic environment (e.g. always diving left for a penalty save).

Noisy Deictic Rules

Noisy Deictic Rules (NDRs) were used as the basis for P-SPOs and were defined in section 2.3.4. NDRs are based on PPDDL, a planning operator representation language for probabilistic domains [99], with restrictions to reduce learning complexity and an addition of *noise* to allow compact modelling of complex environments. NDRs are also an extension of probabilistic STRIPS operators (defined in section 2.3.3) which, independently, formed the background for the stochastic planning operators which were learned by the first form of ASDD presented in [18].

NDRs use two assumptions, which are common to most planning operator definitions, but are amended in the P-SPO definition.

The first is a version of the *frame assumption* (discussed in section 5.5): when an agent takes an action in the environment, anything not explicitly changed stays the same. NDRs relax this assumption, such that operators can have a *noise* result which essentially takes the agent to an undetermined random world state. This mechanism means that the world model provided by the rules will be inaccurate, with some world states generated by the *noise* effect being unreachable. NDRs require the addition of noise because only a single rule can be applied in any time step, with the result that multiple simultaneous changes to the world would require an exponential number of rule outcomes to each rule if there was no recourse to noise.

The blocks-world example demonstrating this need for noise, used in [67], is of a robot knocking over a stack of blocks, which land in a random pile. The possible configurations of this pile are exponential to the number of blocks, and all would need to be modelled as outcomes with associated probabilities for every action if the noise factor was not available. P-SPOs do not suffer from this issue because they can model the situations using multiple single outcome operators. Another example of a use for noise is the action of a strong gust of wind in knocking over a tall stack of blocks. This can be modelled by P-SPOs using environment operators.

NDRs also include a form of *environment rule*. These are simple rules with no conditions that provide background transition probabilities for variables that are never changed by the

operators (for example, the probability that it will start raining while you are trying to move a block). The idea is that these are things outside your control. This is obviously not very expressive in the general case. The lack of conditions means you can only provide the base possibility of transitions (e.g. ignoring the fact that it is a sunny day), and only in the case that a variable is never changed by an action, rather than that it is sometimes changed by an action and sometimes by the environment.

NDRs also include the *outcome assumption*: each action affects the environment in a small number of distinct ways. Each possible effect causes changes to the environment that occur simultaneously as a single outcome. This assumption is, again, required to avoid the need to model environments which would require an exponential number of rules without application of parallel operators. Modelling the action of a *flip-all* action in an *n-coin flip* environment using NDRs requires a rule with 2^n outcomes. This action can be modelled using P-SPOs with n single outcome P-SPOs.

An additional restriction for NDRs is that a well formed (proper) rule set can have a maximum of one applicable rule in any given world state. This *mutual exclusivity of preconditions* between rules has several advantages: there is no need to define *supremacy* between rules; rule application is fast, because the retrieval of a rule which matches the current state precludes the existence of any other matching rule; and it is a prerequisite of the inductive logic programming-based learning method used to acquire the rules from data [67].

The main disadvantage of mutual exclusivity of preconditions is that an exponential number of operators to the number of state variables is required if an output variable is dependent on all conditions. For example, a light that can only be switched on if all n coins in a *coin flip* environment show heads, but will remain off otherwise could be modelled by a single rule, with the frame assumption modelling the *unchanged* state, but if the frame assumption was not made, would require 2^n individual rules to cover all other cases. If the light could randomly turn on irrespective of the agent action, then the frame assumption cannot be used, and NDRs would, again, require 2^n rules.

Deictic References

Deictic references, made popular through demonstration in the Pengi environment [2], are used to describe “the object in front of me” or “the red block over there”. NDRs contain an additional *context* to allow the inclusion of such references. P-SPOs in this research are used to model changes to an agent’s perception of the environment, rather than the environment itself, which can be considered an implicit deictic reference (referring to “the object that can be seen”). The explicit inclusion of deictic references is a subject for future research, but was

not considered in the initial implementation because they are a method of generalisation, and therefore reduce the information content of operator conditions. For example, "the object in front of me" contains less information than "the painted block in front of me". These specific conditions are needed by RVRL for effective learning to take place.

Probabilistic STRIPS Operators (PSOs)

Probabilistic STRIPS operators, PSOs ([48] and expanded in [10] and [32]) were defined in section 2.3.3. In this section we discuss their relation to P-SPOs and NDRs.

The main features for PSOs are shared by NDRs (which use PSOs as a basis):

1) Operators are mutually exclusive (only one operator can be applicable for any given state-action pair).
2) Each operator defines a probability distribution for all changed output variables in all combinations.
3) Output variables which are not mentioned in the rule outcomes remain unchanged.

PSOs can be considered a propositional form of NDRs with the standard frame assumption (no noise or environment rules).

The main advantages of PSOs over non-rule-based formalisms (e.g. influence maps) are that they provide a partial solution to the fame problem, and a compact decision-tree representation can be used for outcome probabilities. These advantages are shared by NDRs.

Other Rule-Based Systems

Reiter's relational STRIPS operators [76] are similar to NDRs but allow conditions to be facts entailed by the database, and include negation by failure. This relaxes the *closed world* assumption implied by the syntax of both NDRs and P-SPOs with the possibility of additional information being added to the database by the operators, rather than all possible facts being either true or false at the outset. Extension to the syntax of P-SPOs to include these properties represents an interesting direction for future research.

Bayesian logic programs, introduced in [43], are a combination of deterministic logic programs with Bayesian networks. These could be adapted to represent dynamic systems and can be learned from data using an inductive logic programming technique [44].

Decision-theoretic logic programs [13] are a recent adaptation of stochastic logic programs [61] to implement decision theoretic reasoning. Currently the formalism does not implement a planning-operator based system, but stochastic logic programs can be adapted to represent

rules of this type, as demonstrated in previous research [18], and can be learned from data using inductive logic programming (ILP) [62][23].

11.2.2 Graphical Models

The standard representations for graphics models of probabilistic temporal networks are based on Bayesian networks. Dynamic Bayesian networks, two-tier Bayesian networks and influence diagrams were discussed in section 2.3. Graphical models are not directly related to probabilistic planning operators, but can be used to model equivalent systems in the prepositional case.

The main advantage of using a Bayesian network based approach is that efficient leaning and inference algorithms have been developed and refined over a number of years, for example: structure learning [32]; learning and inference [63]. See [52] and [46] for in-depth reviews of classic and recent techniques respectively. Both learning and inference are subjects of active ongoing research (e.g. learning [83], inference [36]).

11.2.3 Learning Planning Operators from Experience

Early work on learning planning operators from experience focussed on deterministic planners, for example learning STRIPS like operators for a top-down linear planner [93], and deriving planning rules for a constraint-satisfaction based domain-independent planner [33]. The research presented in this thesis is focussed on probabilistic planning rules. The main research in this area was covered in the background and is discussed further below.

MSDD described in section 3.1.2 has been shown to be an effective method for acquiring probabilistic STRIPS operators (PSOs) from data [65], and can also be used to learn P-SPOs [17]. The ASDD method has been shown to be more efficient than MSDD in the tests presented in section 8, acquiring operators approximately 13 times faster for the *slippery gripper* environment and 15 times faster for the *predator-prey* scenario, with ASDDs being double the speed of ASDD at the expense of additional storage. There was also an indication that the inclusion of an initial filter stage may avoid over-fitting of the data for the slippery gripper problem (section 8.2.1) although this requires further research.

The other main approach discussed in the background chapter was the ILP-based method for discovery of NDRs from data [67]. This method has a restriction that operator conditions must be mutually exclusive, and is not, therefore, directly applicable to P-SPO learning. The need for a full pass over the data set for each additional rule means that the method's speed is likely to be approximately equivalent to that of MSDD for learning an equivalent operator set.

It is interesting to postulate the use of ILP-based techniques for a variable-substitution extension of the ASDD algorithm (discussed in section 11.3.4).

Other recent research has investigated: the derivation of a set of planning operators which are able to bridge the gap between an initial state and a desired goal state using a combination of user-supplied domain knowledge and action traces [77]; and generation of a set of state machines for objects in a domain from state-action traces [20].

11.2.4 State Aggregation Methods

RVRL is a state-aggregation-based approximate dynamic programming method. The main related work in this area is discussed below.

Decision Theoretic Regression

Decision Theoretic Regression [10] uses a decision tree representation of state value, associated with a Dynamic Bayesian Network model of the environment. The method uses structure in the reward function to build a decision tree representation of the value-map which identifies regions of the state-action space whose values are the same. Regressions are made through each action to provide value trees for each available action.

The key difference between the decision theoretic approach and that used by RVRL is that the aggregated states are non-overlapping. This means that each region of the state-action space is covered by exactly one estimate of its value and each backup is essentially a standard Bellman backup, improving a single value estimate based on the successor state-values and probabilities, rather than combinations of estimates. The complexity for decision theoretic regression lies in finding a suitable structural abstraction for the value function. A full comparison with this approach is an interesting avenue for future research.

Explanation based reinforcement learning

Explanation based reinforcement learning (EBRL) [24] is a model-based approach using actions represented by deterministic STRIPS-like operators. The technique combines explanation based learning, which finds sequences of action from an initial state to a goal state, with reinforcement learning's capacity to find optimal policies. Standard explanation based learning (EBL) is prone to finding inefficient solutions to problems, because the first successful sequence of actions which lead to a goal state is kept as the correct solution. EBRL combines the technique with reinforcement learning, by regressing utility values through the operators to the range of states from which the operator can be applied (the operator conditions), and an optimal solution can be found. A requirement of STRIPS operators is that they have mutually exclusive conditions, with the result that each operator represents a unique

partition of the state-action space. EBRL has been extended for actions with stochastic effects by using reward models described by Bayesian networks to partition the state-action space [9]. The technique is similar, with regression through state-action regions encapsulated in mutually exclusive operator conditions. The technique requires a single value to be attributed to each region of the state space and does not, therefore, generalise to multiple concurrent operators (e.g. P-SPOs).

FOALP: First-Order Approximate Linear Programming

The FOALP algorithm [80] is related to RVRL in that it is model-based and uses a fixed set of basis-functions to perform approximate value iteration (AVI) [82]. Each AVI step generates a set of constraints on the contribution that values contained in the basis-functions will make to the global value function. The main difference between FOALP and RVRL is that a linear programming technique is used to solve the constraint satisfaction problem, rather than using weighted state aggregations. This generates an error bound on the error introduced due to approximations. Combining this linear programming approach with RVRL is an interesting avenue for future research.

Relational Reinforcement Learning

Relational Reinforcement Learning is a model-free technique which applies reinforcement learning techniques to first-order representations using inductive logic programming. The seminal technique, RRL, builds a first-order logical decision tree, called a Q-tree, by collecting a trace of state-action pairs with an associated Q-value from each *episode* of experience [29]. An episode is a sequence of action from an initial state to a terminating state. An e-greedy current policy, encapsulated in the Q-tree, is followed in each episode, and any state-action pairs which have not been encountered before are added to the Q-tree, while those that have previously been encountered are used to update the values in the Q-tree using a standard Q-learning update. Experiments in a “blocks world” environment showed that RRL is able to learn a relational representation which can be used in related domains of higher complexity.

The standard RRL method is inefficient because a new Q-tree is regenerated after each episode using all example data generated from previous episodes. This also requires the storage of the previously generated data. These inefficiencies have been addressed in subsequent work, such as: the TG-algorithm, an incremental algorithm which stores statistics in each leaf node of the Q-tree and splits the node when a confidence threshold has been reached [27]; and TRENDI, which is similar to TG, but uses an instance based representation in the leaf-nodes [26].

Standard relational reinforcement learning methods are model-free exact, rather than approximate, solution methods, but are of interest because the policy learning stage of the framework presented in chapter 4 could be achieved using either batch (for BatchModelQ) or incremental (for Dyna-Q) versions of the algorithm. Recent work has extended the RRL approach by integrating a model learner (Bayesian network based) with a relational regression tree learner (built incrementally) [21]. This demonstrates a model-acquisition technique that could be adapted to use RVRL.

11.3 Future Work

The following sections outline directions for future work and some partially explored work in the thesis which warrants further investigation.

11.3.1 Use of Approximate Dynamic Programming Update Techniques

The RVRL system presented in this thesis used a random selection technique to select states and actions for updates. Previous work has shown that following a policy and picking successor states for updates from the model can produce an accurate model under some circumstances. The method was not presented in depth in this thesis because it was found to be sensitive to the experimental conditions, such as discount rate, and could fall into local minima. An exploration of RVRL based on these updates can be found in the author's previous work on RVRL [16]. These issues are also explored in the context of approximate dynamic programming under on-policy vs. off-policy updates [73].

If the policy is followed when selecting states to update, then the estimates for poor states becomes inflated because the operators approximate both good states (which are visited) and poor states (which are not). If the policy is not followed when selected states to update, then estimates for good states can be depressed by poor states which would never be visited by a greedy policy.

A further issue occurs in environments in which reward tends to accumulate. States which are rarely visited will tend to have lower value estimates because they have not been visited often. This effect is alleviated to some extent by the use of McClain updates (section 3.2.4).

A full experimental investigation would improve understanding of the issues presented by on-policy vs. off-policy updates. An interesting area would be investigation of a mixed strategy, in which on policy updates are followed initially in order to provide information on the structure of the reward function, followed by randomised state-action selection to refine values towards an optimal policy.

11.3.2 Parallelisation of ASDD

The increased applicability of association rule mining is due, in part, to the application of parallelisation techniques for spread of the search over multiple processors. A key recent advance has been the MapReduce technique [22]. The technique has been applied to the Apriori algorithm [50], and it is reasonable to assume that this technique could, with suitable modification, be applied to the Apriori-based ASDD algorithm.

11.3.3 In-line ASDD

The separation of the experience gathering and learning stages is required due to the batch nature of the rule learning algorithm used. This type of approach is common if rules are to be built from experience [51], and has an advantage in that the environment model is distinct from the goals, with the results that the model is still valid in the face of a changing reward function. The framework presented in chapter 4 could be adapted to incorporate incremental operator learning and operator-value updates to give an essentially Dyna-Q-like approach. An incremental association rule learning algorithm [41][14] could extend the Apriori Stochastic Dependency Detection method (ASDD) defined in chapter 6 to learn operators in this way. Incremental algorithms store statistics at the border between accepted associations and those with less that minimum support (or in the case of ASDD below significance threshold). This threshold can be breached with further data, allowing the generation of child rules which then become the new threshold, gathering statistics in the same way.

11.3.4 Variable substitution in ASDD

The P-SPO definition includes the use of variables which can form parameters for actions and can be substituted within action conditions. The ASDD learner presented does not, however, address learning rules of this type. The substitution of variables was not addressed because it introduces a further level of complexity and is unlikely to improve the performance of the RVRL algorithm. As shown in the slippery gripper experiments, reduction in information in the rules can mean that RVRL is lacking information on which to base state-value aggregations (e.g. the *dryer* operator issue described in section 10.1.3). Increased generalisation by variable substitution would cause the rules to have less information and thus compound the issues for RVRL.

From a modelling point of view, however, the use of variable substitutions could improve the model by grouping together similar actions (e.g. *move(X): see(X, wall)* → *{1.0: see(X, wall)}*) allowing improved performance of the model in unseen states.

A simple method for performing these substitutions post-learning, from the operator set created by ASDD, is to substitute a variable in one of the rules and remove all rules for which this new rule provides an outcome which is not significantly different. This can be achieved using the G-statistic in a similar way to the generalisations through rules with fewer conditions in the ASDD algorithm.

For example, a new rule can be generated by substituting a variable for *north* in the rule:

move(north), see(north, wall) → *{1.0: see(north, wall)}.*

Giving:

move(X), see(X, wall) → *{1.0: see(X, wall)}.*

All rules which match these conditions can then be tested for significant difference with the new rule. The other rules which match this new rule are:

move(south, see(south, wall) → *{1.0: see(south, wall)}*

move(east), see(east, wall) → *{1.0: see(east, wall)}*

move(west), see(west, wall) → *{1.0: see(west, wall)}*

Each of these rules is generalised by the new rule and can, therefore, be removed.

11.3.5 Learning P-SPOs with Dependencies in Outputs

The predator-prey scenario demonstrated the modelling weakness of a rule system which does not make provision for dependencies between outputs. Again, the P-SPO operator definition provides a mechanism for including dependencies between outputs, but the ASDD algorithm does not learn these dependencies in its current form.

Two possible extensions to ASDD to learn operators of this type are:

1) *Combine operators with the same conditions but different effects:* in this case, the extra effect can be treated as a condition to each of the individual outcomes and a significance test used to discover if this extra condition has a significant effect on the probability of the outcome. If it does, then the combined rule should be used in place of both the individual rules.

2) *Extend ASDD to include outcomes as conditions:* ASDD is based on the Apriori algorithm for building association rules from data. Apriori makes no distinction between cause and effect, only finding items occurring together regularly. ASDD

could also ignore the distinctions between outcomes and conditions when learning rules. Adjustments would need to be made to the *filter* and *aprioriFilter* processes.

11.3.6 Variable Percept Size

The P-SPO operator definition does not include provision for add/delete operators in the environment. Rather, all conditions in the environment are either present or not present in the environment at all times. An interesting extension to the syntax would be to include add/delete outcomes to the operator definition and to explore modelling these operators using an extension of ASDD.

11.3.7 Augmenting Operators with High Variance in RVRL

The major limitation of the RVRL approach is the lack of information in some P-SPOs. A good example of this is the *dryer* action. The action is fairly simple. The only condition required by the operator is whether the gripper was wet when the action was applied, while the value of taking the action is dependent upon the whether the agent is currently holding the block, and whether the block is painted.

When examining the variance of the P-SPO, it was noted that the rule's value is largely dependent on whether the block is currently painted. This operator has a relatively low variance, because the following state (block painted) is valuable, but the accuracy of the estimate could be improved greatly by knowledge of whether the block was held.

An area of future research would be to assess the variance of applicable operators for each state-action combination. If all operators are found to variance above a threshold for a particular state-action, then the utility estimate for this state-action is unstable. In such cases, operator conditions for the lowest variance operator could be augmented with additional conditions present in other operators to create finer grained aggregations. There is clearly a trade-off involved with this approach in that additional conditions will eventually lead to the "curse of dimensionality" that RVRL aims to avoid.

11.4 Summary

This chapter discussed the results of this research in relation to the initial objectives, concluding that these objectives have been largely fulfilled, and that the approach is a promising development in the field of approximate dynamic programming. Comparisons were made with related work in model definition, model learning, and state-aggregation-based reinforcement learning.

Some issues with the ASDD operator learning algorithm were highlighted, including limitations in learning operator sets with conditional outputs and the substitution of variables in the operator definitions. Limitations to the RVRL algorithm when presented with a lack of information present in an environment which can be represented by a compact operator set were also highlighted. Each of these limitations has been addressed in the future work section, with some suggestions for approaches which could resolve these issues and widen the applicability of the research.

Appendix A: Definitions and Symbols

This section defines basic probability theory principles and the G statistic test of non-independence.

A.1 Basic Concepts in Probability Theory

Probabilities in this thesis are treated according to the Bayesian interpretation, in which probabilities encode a degree of belief about events in the world and data provides evidence for the degree of belief. Propositions (sentences with true or false output) are assigned a degree of belief and can be combined and manipulated according to the rules of probability calculus (definition adapted from [69]).

There are three basic axioms of the probability calculus:

$$0 \le P(A) \le 1, \tag{12.1}$$

$$P(\text{definite proposition}) = 1 \tag{12.2}$$

$$P(A \text{ or } B) = P(A) + P(B) \text{ if } A \text{ and } B \text{ are mutually exclusive.} \tag{12.3}$$

P(A) is the probability that a proposition (sentence) *A* will be true. The third axiom above states that the probability of a any of a set of events occurring is the sum of the probabilities of each of the events occurring if the events cannot occur simultaneously (they are mutually exclusive). An example of mutually exclusive events is the probability that a coin will show heads or tails.

If two events are not mutually exclusive and are independent (the probability of one is not influenced by the probability of the other), then the probability of both occurring together is:

$$P(A \text{ and } B) = P(A) \times P(B) \text{ if } A \text{ and } B \text{ are independent.} \tag{12.4}$$

Events are independent if the probability of *A* given that *B* has already occurred is equal to the probability of *A* (the probability remains unchanged). Written:

$$P(A \mid B) = P(A). \tag{12.5}$$

If two events are not mutually exclusive and are not independent, then the probability of *A* is dependent on the probability of *A* given *B* and *A* given ¬*B*.

$$P(A) = P(A \mid B)P(B) + P(A \mid \neg B)P(\neg B) \tag{12.6}$$

More generally, if *B* is a variable, which can take any one of a number of *n* values from B_1 to B_n then:

$$P(A) = \sum_i P(A|B_i)P(B_i) \tag{12.7}$$

A.2 The G statistic

The G statistic [97] is a statistical test of non-independence, computed for a 2×2 contingency table. The test is used for filtering and pruning of planning operators because it is valid when the expected probabilities are small, or the number of observations in the data is small (as can be the case in both the MSDD and ASDD algorithms).

Table 11-1: 2×2 contingency table for the co-occurrence of x and y.

	y	***¬y***
x	n_1	n_2
¬x	n_3	n_4

Table 11-1 gives a contingency table for the co-occurrence of two variables *x* and *y*. The numbers in the tables are:

- n_1 = the number of times that *x* and *y* occur together in the data.
- n_2 = the number of times that *x* occurred but *y* did not occur.
- n_3 = the number of times that *x* did not occur and *y* occurred.
- n_4 = the number of times that neither *x* nor *y* occurred.

Additionally:

- r_1 = the sum of the first row ($n_1 + n_2$): the number of times that *x* occurred in the data.
- r_2 = the sum of the first column (n_1+n_3): the number of times that *y* occurred in the data.
- t = total ($n_1+n_2+n_3+n_4$): the total number of observations in the data set.

The G statistic is computed from the table above as:

$$G = 2\sum_{i=1}^{4} n_i \log(n_i / \hat{n}_i) \tag{12.8}$$

The numbers $\hat{n}_i$ are the expected values of each n_i if *x* and *y* are independent. This can be calculated from the row totals and table total. For example, if *x* and *y* are independent

variables then the expected probability of both x and y occurring together is the probability of x, $pr(x)$, multiplied by the probability of y, $pr(y)$. $pr(x) = r_1/t$. $pr(y) = c_1/t$. The expected value of n_1 (observed co-occurrence of x and y in the data) is the total number of observations multiplied by $pr(y) \times pr(x) = t \times r_1/t \times c_1/t = r_1 \times c_1/t$.

Expected values of each of the other entries in the table can be calculated by an equivalent method.

A low value of G is an indication of independence in the data, while a high value of G indicates non-independence. Table 11-2 gives the probability of making an error in rejecting the null hypothesis (that the two variables are independent). The values in this table are equivalent to those for the χ^2 distribution with one degree of freedom.

Table 11-2: Significance levels given by G vales for the G statistic test

Significance Level	0.5	0.10	0.05	0.02	0.01	0.001
G Value	0.455	2.706	3.841	5.412	6.635	10.827

The G statistic is used in to filter specific versions of a rule with general versions of the rule. In other words, both rules refer to the same outcome, with the more general rule being applicable to more of the data set. If r_1 is a more general version of r_2, then the data set of observations can be restricted to the observations matching the more general rule. This method for rule pruning was introduced by Oates & Cohen [65].

In this case:

- n_1 = the number of times the outcome y was observed after the general conditions x.
- n_2 = the number of times the outcome y was not observed after the general conditions x.
- n_3 = the number of times the outcome y was observed after the specific conditions (equivalent to $\neg x$).
- n_4 = the number of times the outcome y was not observed after the specific conditions (equivalent to $\neg x$).

Any rule fits into the pattern as:

$$\begin{aligned} &\text{General rule is written} : x \rightarrow y \\ &\text{Specific rule is written} : \neg x \rightarrow y \end{aligned} \tag{12.9}$$

The pseudo-code for the G statistic is given below:

```
GStatistic (d1, d2)
   n1 = d1.sc;
   n2 = d1.bs – d1.sc;
   n3 = d2.sc;
   n4 = d2.bs – d2.sc;

   r1 = n1 + n2;
   r2 = n3 + n4;
   c1 = n1 + n3;
   c2 = n2 + n4;
   t = r1 + r2;

   //check for both rules being equal or having 1.0 prob
   //remove the specific one as it is covered by the general one
   if (((n1 == n3) and (n2 == n4)) or
       ((n2 + n4) == 0)
      return 0;

   return 2.0f *
   (n1*log((n1*t)/(r1*c1)) +
    n2*log((n2*t)/(r1*c2)) +
    n3*log((n3*t)/(r2*c1)) +
    n4*log((n4*t)/(r2*c2)));
```

Algorithm 11-1: G statistic. d1 = general rule, d2 = specific rule, sc is the support count for the rule in the observed data, bs is the support count for the body (conditions) of the rule in the observed data. The algorithm returns the G statistic measure of non-independence between d1 and d2.

The G statistic is used to perform post-pruning of the data in the MSDD and ASDD algorithms presented. The values can also be used to perform pre-pruning of the data by finding an upper bound on the value of G for a general rule versus its (more-specific) children. This method is used in the original MSDD algorithm [64]. ASDD performs a pruning based on a fixed high valued G-statistic of a node's decedents in the Apriori-Filter step (see section 6.4.9). The pre-pruning is based on rules with three less conditions.

Appendix B: Program Code and Output

The full code-base used for the experiments in this thesis is available from:

www.soi.city.ac.uk/eu779/PhDThesisCode/PhDThesisCode.zip

The raw data output and saved P-SPO sets for the experiments are available from:

www.soi.city.ac.uk/eu779/PhDThesisResults/PhDThesisResults.zip

The flowing sections contain detailed pseudo-code for two of the algorithms included in the main thesis.

B.1 Apply By Sample

Apply be sample function takes a percept P and a P-SPO as input. The values of the input percept are changed by taking a sample output from the P-SPO. Each outcome in the sampled output set replaces the matching feature in the percept.

```
applyBySample(P, PSPO)
   //generate a random number from 0 to 1
   rouletteVal = random(0.0, 1.0);
   //sum the probabilities of the outputs until
   //total is greater than the sample position
   samplePos = 0;
   On = first(PSO.PO);
   Pn = first(PSO.PP);
   do {
      samplePos += Pn;
      On = next(On);
      Pn = next(Pn);
   } (while (samplePos < rouletteVal)
   //Replace all perceptual features in the input percept
   //with the features from the output set
   for (PerceptFeatureItt ∈ On) {
      replaceAtConflicting(P, PerceptFeatureItt);
   }
```

Algorithm 11-2: applyBySample(P, PSPO). P = initial percept, PSPO = the P-SPO to apply. The algorithm returns a single percept with one of the outcomes of the P-SPO applied using probabilistic sampling.

B.2 Apply All Outcomes

The `applyAllOutcomes` function of a P-SPO takes a percept P a P-SPO and an initial probability as input. It returns a set of percepts and associated probabilities after applying each outcome from an outcome set in turn to the percept. The probability is the input probability multiplied by the probability of the outcome set. The input probability will be 1.0 if this is the first outcome set applied, and the combined probabilities otherwise.

```
applyAllOutcomes(P, PSPO, Prob)
   OutputSet = {};
   Pn = first(PSO.P_P);
   for (On ∈ PSO.P_O) {
      NextOutput = P;
      OutputProb = Prob × Pn;
      for (FeatureItt ∈ On)
         replaceAtConflict(NextOutput, FeatureItt);
      add(OutputSet, {NextOutput,OutputProb};
      Pn = next(Pn);
   }
   return OutputSet;
```

Algorithm 11-3: applyAllOutcomes(P, PSPO, Prob). P = input percept. PSPO = P-SPO to apply. Prob = probability associated with initial percepts. The function applies the outcomes of each of the outcome-sets of the P-SPO in turn to the percept, and returns a set of percepts with associated probabilities.

12. Glossary

Autonomous Agent: a decision-making entity situated in an environment or world. It has a number of actions it can carry out, a method of perceiving its environment and makes decision as to which of the available actions it will select.

Batch process: execution of a series of programs to process data without additional input.

Conditional Independence: two events are conditionally independent if the outcome of each is independent given that a third event (or set of events) has occurred.

Continuous Task: a task with no terminating states and therefore an infinite number of stages.

Deictic References: a reference to an object or attribute from the perspective of a given context. E.g. the object that is in front of the agent.

Dependent Operator Outcomes: operator outcomes which are dependent on the values of features of the successor state.

Distribution Model: a function returning a set of states (or percepts) and associated probabilities in response to an input state (or percept) and action.

e-Greedy action: an action determined by the current policy or, with a defined probability, a random action.

Environment Model: a function describing the transition between environment states in response to actions.

Environment Modelling Agent: an agent which builds a model of its environment through interaction.

Episodic Task: a task with defined terminating states. The trajectory from a start state to an end state is an episode.

Factored State Model: a state transition model defined in terms of a set of environment features.

Finite-Horizon Problem: see *episodic task.*

Frame Assumption: a simplifying assumption stating that all features of an environment not explicitly changed by the outcome set of a planning operator remain unchanged.

G Statistic: a statistical test of non-independence.

Independent Operator Outcomes: operator outcomes which are dependent only on conditions and actions.

Infinite-Horizon Problem: see *continuous task.*

Markov Assumption: a simplifying assumption that the current state of the world contains sufficient information to predict the probability of the following state.

Markov Model: a model defining a set of probability distributions describing the transition between current and next states depending on a set of past states.

Noisy Deictic Rules (NDRs): a relational extension of PSOs to include deictic references, variables in the head and body of the rule, and noise outcomes.

Non-Player Character (NPC): the name often used synonymously with autonomous agents in the context of computer games. NPCs can be adversaries, helpers or background characters.

On-Line Processing: execution of a series of programs to process data during which additional input may be provided.

Operator Context: a set of conditions which determine when a P-SPO can be applied.

Operator Precedence: determines the P-SPO to be applied in generating a successor state (or percept) if two or more operators are applicable to the current state (or percept) and have conflicting outcomes.

Parallel Stochastic Planning Operators (P-SPOs): planning operators with probabilistic outcome sets which can be applied in parallel to describe expected changes to an environment in response to an action.

Percept: the input an agent is currently receiving from its environment. This can be a direct representation of the output of sensors, or can be pre-processed information from sensors.

Perceptual Data Item (PDI): a percept, action and successor percept.

Perceptual Model: a model defining the transitions between agent percepts in response to actions.

Plan: a deterministic set of actions which lead an agent from a current state to a goal state.

Planning: the process by which an agent creates a plan.

Planning Operator: an expression describing the expected changes to an agent's environment in response to an action, given certain preconditions.

Policy: a universal plan mapping each possible state to probabilities of selecting each possible action.

Probabilistic STRIPS operators (PSOs): a probabilistic extension of STRIPS operators allowing multiple outcome-sets with associated probabilities.

P-SPO Support: the proportion of the PDIs in a data set that contain the conditions, action and a single outcome-set of a given P-SPO.

Support Count: the number of PDIs in a data set containing the conditions, action and a single outcome-set of a given P-SPO.

Q-Value: an estimate of expected discounted future reward for taking a given action in a given state.

Reinforcement Learning: an on-line process by which Q-values are updated by feeding back discounted rewards from a successor state to a current state.

Stochastic Environment: an environment for which the transition between states is non-deterministic. The subsequent state is determined by predictable actions and a random element.

STRIPS operator: a deterministic planning operator with action, preconditions and effects. The effects define the changes to the environment if the action is taken under the conditions.

Tabular Model: a state transition model represented as a table with two columns: the current state and the next state.

Utility: expected total discounted future rewards.

13. References

[1] Agrawal, R. and Srikant, R. (1994) "Fast Algorithms for Mining Association Rules", *Proc. of the 20th Int. Conf. Very Large Data Bases*, *Santiago, Chile*, pp 487-499.

[2] Agre, P. and Chapman, D. (1987) "Pengi: An Implementation of a Theory of Activity", *Proc. of the Sixth Nat. Conf. on Artificial Intelligence (AAAI-87)*, *Seattle, Washington, USA*, pp 268-272.

[3] Bartak, R., Salido, M. A. and Rossi, F. (2010) "New Trends in Constraint Satisfaction, Planning and Scheduling: a Survey", *The Knowledge Engineering Review*, Cambridge University Press, pp 249-279.

[4] Bhatnagar, V., Gupta, A. and Kumar, N. (2009) "Algorithms for Association Rule Mining", *Encyclopaedia of Artificial Intelligence*. IGI Global, pp 76-84.

[5] Bellman, R. E. (1957) "Dynamic Programming". Princeton University Press.

[6] Bellman, R. E. (1961) "Adaptive Control Processes: A Guided Tour". Princeton University Press.

[7] Boutilier, C., Dean, T. and Hanks, S. (1999) "Decision-Theoretic Planning: Structural Assumptions and Computational Leverage", *Journal of Artificial Intelligence Research, 11*, pp 1-94.

[8] Boutilier, C., Dearden, R. and Goldszmidt, M. (1995) "Exploiting Structure in Policy Construction", *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence (IJCAI-95)*, *Montreal, Canada*, pp 1104-1111.

[9] Boutilier, C., Reiter, R. and Price, B. (2001) "Symbolic Dynamic Programming for First Order MDPS", *Proc. of the 17th Int. Joint Conference on Artificial Intelligence (IJCAI-01)*, *Seattle, Washington, USA,* pp 690-697.

[10] Boutilier, C., Dearden, R. and Goldszmidt, M. (2002) "Stochastic Dynamic Programming with Factored Representations", *Artificial Intelligence*, *121 (1-2),* pp 49-107.

[11] Brooks, R. A. (1991) "Intelligence without Representation". *Artificial Intelligence, 47*, pp 139-159.

[12] Buckland, M. (2004) "Programming Game AI by Example". Wordware Publishing Inc.

[13] Chen, J. and Muggleton, S. (2009) "Decision-Theoretic Logic Programs", *Proc. 19th Intl. Conf. on Inductive Logic Programming (ILP 2009), Leuven, Belgium*.

[14] Cheung, D. W., Han, J., Ng, V. and Wong, C. Y. (1996) "Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique", *Proc. 1996 Int. Conf. Data Engineering (ICDE 96)*, *New Orleans, Louisiana*, pp 106-114.

[15] Child, C., Stathis, K. and Garcez, A. (2007), "Learning to Act with RVRL Agents", *Proc. 14th RCRA Workshop, Experimental Evaluation for Solving Problems with Combinatorial Explosion*, *Rome, Italy*.

[16] Child, C. and Stathis, K. (2006) "Rule Value Reinforcement Learning for Cognitive Agents", *Proc. 5th Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS-06), Hakodate, Japan*.

[17] Child, C. and Stathis, K. (2005) "SMART (Stochastic Model Acquisition with ReinforcemenT) Learning Agents: A Preliminary Report", *Adaptive Agents and Multi-Agent Systems II: Adaptation and Multi-Agent Learning*, Kudenko, D., Kazakov, D. and Alonso, E. (eds.), LNAI 3394. Springer, pp 73-87.

[18] Child, C. and Stathis, K. (2004) "The Apriori Stochastic Dependency Detection (ASDD) Algorithm for Learning Stochastic Logic Rules", *Computation Logic in Multi-Agent Systems: 4th Int. Workshop, CLIMA IV,* Dix, J. and Leiter, J. (eds.), LNAI 3259. Springer, pp 234-249.

[19] Coulom, R. (2006) "Efficient Selectivity of Backup Operators in Monte-Carlo Tree Search", *Proc. 5th Int. Conf. on Computers and Games (CG 2006)*, *Turin, Italy,* Ciancarini, P. and van den Herik, H.J. (eds.), pp 72-83.

[20] Cresswell, S. and Gregory, P. (2011) "Generalised Domain Model Acquisition from Action Traces", *Proc. 21st Int. Conf. on Automated Planning and Scheduling (ICAPS 2011)*, pp 42-49.

[21] Croonenborghs, T. (2009) "Model-Assisted Approaches for Relational Reinforcement Learning", *Ph.D. Thesis*, Katholieke Universiteit Leuven.

[22] Dean, J. and Ghemawat, S. (2004) "MapReduce: Simplified Data Processing on Large Clusters", *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI 04),* pp 1-13.

[23] De Raedt, L. and Kersting, K. (2004) "Probabilistic Logic Learning", *ACM-SIGKDD Explorations, Special issue on Multi-Relational Data Mining, Vol. 5*, pp 31-48.

[24] Dietterich, T. G. and Flann, N. S. (1997) "Explanation-Based Learning and Reinforcement Learning: A Unified View", *Machine Learning, 28,* pp 169-214.

[25] Draper, D., Hanks, S. and Weld, D. (1994) "Probabilistic Planning with Information Gathering and Contingent Execution", *Proc. 2nd Int. Conf on Artificial Intelligence Planning Systems (AIPS-94)*, pp 31-36.

[26] Driessens, K., and Dzeroski, S. (2005) "Combining Model-Based and Instance-Based Learning for First-Order Regression", *Proc. 22nd Int. Conf. on Machine Learning (ICML 05)*, pp 193-200.

[27] Dreissens, K., Ramon, J. and Blockeel, H. (2001) "Speeding up Relational Reinforcement Learning Through the use of an Incremental First Order Decision Tree Learner". *Proc. of the 13th European Conference on Machine Learning (ECML 01)*, pp 97-108.

[28] Drescher, G. L. (1991) "Made-Up Minds: A Constructivist Approach to Artificial Intelligence". The MIT Press.

[29] Dzeroski, S., De Raedt, L. and Driessens, K. (2001) "Relational Reinforcement Learning", *Machine Learning, 43 (1-2)*, pp 7-52.

[30] Ferber, J. (1999) "Multi-Agent Systems: An introduction to Distributed Artificial Intelligence". Addison-Wesley.

[31] Fikes, R. E. and Nilsson, N. J. (1971) "STRIPS: a New Approach to the Application of Theorem Proving to Problem-Solving". *Artificial Intelligence, 2 (3-4)*, pp 189-208.

[32] Friedman, N. and Goldszmidt, M. (1998). "Learning Bayesian Networks with Local Structure", *Learning in Graphical Models (Adaptive Computation and Machine Learning),* Jordan, M. I. (Ed.). MIT Press, pp 421-459.

[33] Fox, M. and Long, D. (1998) "The Automatic Inference of State Invariants in TIM", *Journal of Artificial Intelligence Research*, *9*, pp 367-421.

[34] Hanks, S. (1990) "Projecting Plans for Uncertain Worlds", *Ph.D. thesis 756*, Yale University, Department of Computer Science, New Haven, CT.

[35] Hart, P., Nilsson, N. J. and Raphael, B. (1968) "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on System Science and Cybernetics SSC4, 4(2)*, pp 100-107.

[36] Gatti, E., (2011) "Graphical Model for Continuous Time Inference and Decision Making", *Ph.D. Thesis*, Universita Degli Studi Di Milano.

[37] Goertzel, B. Ikle, M., Goertzel, I. F. and Heliakka, A. (2008) "Probabilistic Logic Networks: A Comprehensive Framework for Uncertain Inference", *Springer*.

[38] Gregory, J., Lander. J. and Whiting, M. (2009) “Game Engine Architecture”, A.K. Peters/ CRC Press.

[39] Grzes, M. and Kudenko, D. (2008) “An Empirical Analysis of the Impact of Prioritised Sweeping on the DynaQ’s Performance”, *Proc. 9th Int. Conf. on Artificial Intelligence and Soft Computing (ICAISC-08)*, pp 1041-1051.

[40] Guerin, F. (2011) "Learning like a Baby: A Survey of AI approaches", *The Knowledge Engineering Review* (accepted to appear).

[41] Hidber, C. (1999) “Online Association Rule Mining”. *Proc. of the 1999 ACM SIGMOD Int. Conf. on Management of Data, Philadelphia, Pennsylvania, USA,* pp 145-156.

[42] Hipp, J., Gunter, U. and Nakhaeizadeh, G., (2000) “Algorithms for Association Rule Mining: a General Survey and Comparison”, *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2000), Boston, MA, USA*, pp 58-64.

[43] Kirsting, K. and De Raedt, L. (2001) “Bayesian Logic Programs”, Technical Report No. 151, Institute of Computer Science, University of Freiburg, Germany.

[44] Kirsting, K. and De Raedt, L. (2008) “Basic Principles of Learning Bayesian Logic Programs”, *Probabilistic Inductive Logic Programming*, Lecture Notes in Computer Science, 4911/2008. Springer.

[45] Kisynski, J. and Poole, D. (2009) “Lifted Aggregation in Directed First-Order Probabilistic Models”, *Proc. 21st Int. Joint Conf. on Artificial Intelligence (IJCAI 2009),* pp 1922-1929.

[46] Koller, D. and Friedman, N. (2009) “Probabilistic Graphical Models: Principles and Techniques”. The MIT Press.

[47] Kowalsi, R. and Sergot, M. (1986) “A Logic-Based Calculus of Events”, *New Generation Computing 4*, pp 67-95.

[48] Kushmerick, N., Hanks, S., and Weld, D. S. (1995) “An Algorithm for Probabilistic Planning”, *Artificial Intelligence*, *76* (1–2), pp 239–286.

[49] Jennings, N. R. and Wooldridge, M. (1995) “Intelligent Agents, Theory and Practice”, *The Knowledge Engineering Review, 10 (2),* pp115-152.

[50] Jiang, W., Ravi, V. and Agrawal, G. (2010) “A Map-Reduce System with an Alternative API for Multi-Core Environments”, *Proc. 10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing (CCGRID 2010).*

[51] Jimenez, S. (2007) "Learning Actions Success Patterns from Execution", *Doctoral Consortium*, *17th International Conference on Automated Planning and Scheduling (ICAPS-07).*

[52] Jordan, M. I., (1998) "Learning in Graphical Models". MIT Press.

[53] Lang, T. and Toussaint, M. (2009) "Approximate Inference for Planning in Stochastic Relational Worlds", *Proc. 26th Intl. Conf. on Machine Learning (ICML 2009)*, pp 585-592.

[54] Lang, T. and Toussaint, M. (2010) "Planning with Noisy Probabilistic Relational Rules", *Journal of Artificial Intelligence Research, 39,* pp 1-49.

[55] Lavrac, N. and Dzeroski, S. (1994) "Inductive Logic Programming Techniques and Applications." Ellis Horwood.

[56] Lesperance, Y., Levesque, H. J. and Reiter, R. (1999) "A Situation Calculus Approach to Modelling and Programming Agents", *Foundations of Rational Agency*, Rao, A. and Wooldridge, M. (eds.), pp 275-299. Kluwer.

[57] McCarthy, J. and Hayes, P. J. (1969) "Some Philosophical Problems from the Standpoint of Artificial Intelligence", *Machine Intelligence*, *4*, pp 463-502.

[58] McClain, J. (1974) "Dynamics of Exponential Smoothing with Trend and Seasonal Terms", *Management Science 20*, pp 1300-1304.

[59] McShaffry, M. (2009) "Game Coding Complete: Third Edition". Cengage Learning.

[60] Moore, A. and Atkeson, C. (1993) "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time", *Machine Learning, 13,* pp 103-130.

[61] Muggleton, S. H. (1996) "Stochastic Logic Programs", *Advances in Inductive Logic Programming*, de Raedt, L. (ed.), IOS Press, pp 254-264.

[62] Muggleton, S. H. (2000) "Learning Stochastic Logic Programs", *Proc. of the AAAI Workshop on Learning Statistical Models from Relational Data*, Getoor, L. and Jensen, D. (eds.), pp 36-41.

[63] Murphy, K. P. (2002) "Dynamic Bayesian Networks: Representation, Inference and Learning". *Ph.D. Thesis*, University of California, Berkeley.

[64] Oates, T. and Cohen, P. R. (1996) "Searching for Structure in Multiple Streams of Data", *Proc. 13th Int. Conf. on Machine Learning (ICML 96),* pp 346-354.

[65] Oates, T. and Cohen, P. R. (1996). "Learning Planning Operators with Conditional and Probabilistic Effects". *Proc. AAAI Spring Symposium on Planning with Incomplete Information for Robot Problems*, pp 86-94.

[66] Pasula, H. M, Zettlemoyer, L. S. and Kaelbling, L. P. (2004) "Learning Probabilistic Relational Planning Rules", *Proc. 14th Int. Conf. on Automated Planning and Scheduling (ICAPS-04)*, pp 73-82.

[67] Pasula, H. M, Zettlemoyer, L. S. and Kaelbling, L. P. (2007) "Learning Symbolic Models of Stochastic Domains", *Journal of Artificial Intelligence Research, 29*, pp 309-352.

[68] Pearl, J. (1998) "Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference". Morgan Kaufmann, San Mateo.

[69] Pearl, J. (2000) "Causality: Models, Reasoning and Inference". Cambridge University Press.

[70] Poole, D. (1995) "Logic Programming for Robot Control", *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence* (*IJCAI-95),* pp 150-157.

[71] Poole, D. (1997) "The Independent Choice Logic for Modelling Multiple Agents Under Uncertainty", *Artificial Intelligence*, *94 (1-2), Special Issue on Economic Principles of Multi-agent Systems*, pp 5-56.

[72] Poole, D. (2008) "The Independent Choice Logic and Beyond", *Probabilistic Inductive Logic Programming*, De Raedt, L., Frasconi, P., Kersting, K. and Muggleton, S.H. (eds.), pp 222-243.

[73] Powell, W. B. (2007) "Approximate Dynamic Programming: Solving the Curses of Dimensionality". Wiley Inter-Science.

[74] Reiter, R. (1978) "On Closed World Data Bases", *Logic and Data Bases*, Gallaire, H. and Minker, J. (eds.). Plenum Press, New York, pp 55-76.

[75] Reiter, R. (1980) "A logic for default reasoning", *Artificial Intelligence, 13*, p81-132.

[76] Reiter, R. (2001) "Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems". The MIT Press.

[77] Richardson, N. E. (2008) "An Operator Induction Tool Supporting Knowledge Engineering in Planning", *Ph.D. Thesis*, University of Huddersfield.

[78] Salido, M. (2010) "Introduction to Planning, Scheduling and Constraint Satisfaction", *Journal of Intelligent Manufacturing, 21 (1),* pp 1-4.

[79] Sanner, S. P. (2008) "First-Order Decision Theoretic Planning in Structured Relational Environments", *Ph.D. Thesis*, Graduate Department of Computer Science, University of Toronto.

[80] Sanner, S. P. and Boutilier, C. (2005) "Approximate Linear Programming for First-Order MDPs", *Proc. 21st Conf. on Uncertainty in Artificial Intelligence (UAI-2005)*, pp 509-517.

[81] Savasere, A., Omiecinski, E., Navathe, S. (1995) "An Efficient Algorithm for Mining Association Rules in Large Databases", *Proc. 21st Conf. on Very Large Databases, (VLDB 95), Zurich, Switzerland*, pp 432-443.

[82] Schuurmans, D., Patrascu, R. (2001) "Direct Value-Approximation for Factored MDPs", *Proc. 15th Annual Conf. on Neural Information Processing Systems* (*NIPS 2001)*, pp 1579-1586.

[83] Scutari, M. (2010) "Learning Bayesian Networks with the bnlearn R Package", *Journal of Statistical Software*, *35(3),* pp 1-22

[84] Shen, W. (1993) "Discovery as Autonomous Learning from the Environment", *Machine Learning, 12,* pp 143-165.

[85] Stathis, K., Child, C., Lu, W. and Lekeas, G. K. (2002) "Agents and Environments", *Technical Report IST32530/CITY/005/DN/I/a1*, SOCS Consortium.

[86] Stathis, K., Kakas, A. C., Lu, W. Demetriou, N., Edriss, U. and Bracciali, A. (2004) "PROSOCS: a Platform for Programming Software Agents in Computational Logic", *Proc. of the 4th Int. Symposium: from Agent Theory to Implementation (AT2AI-4)*, Muller, J. and Petta, P. (eds.), pp 523-528.

[87] Sutton, R. S., and Barto, A. G. (1998) "Reinforcement Learning: An Introduction". The MIT Press.

[88] Sutton, R. S. (1990) "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming", *Proc. of the 7th Int. Conf. on Machine Learning (ICML 90)*, pp 216–224.

[89] Sutton, R. S. (1990) "Time-Derivative Models of Pavlovian Reinforcement", *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, Gabriel, M. and Moore, J. (eds.), Cambridge MIT Press, pp 497-537.

[90] Varsy, R. (2002) "Extending Planning and Learning through Reinterpretation of World Model", *M.Sc. Thesis*, City University London.

[91] Tesauro, G. J. (1994) "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play." *Neural Computation, 6 (2),* pp 215-219.

[92] Turing, A. M. (1950) "Computing Machinery and Intelligence", *Mind, 49*, pp 433-460.

[93] Wang, X. "Planning while Learning Operators", *in Proc. of the 3rd Int. Conf. on Artificial Intelligence Planning Systems (AIPS-96)*, AAAI press, pp 229-236

[94] Watkins, C. J. C. H. (1989) "Learning from Delayed Rewards." *Ph.D. Thesis*, Cambridge University.

[95] Watkins, C. J. C. H. and Dayan, P. (1992) "Technical Note: Q-Learning." *Machine Learning, 8*, pp 279-292.

[96] Webb, G. I. (2007) "Discovering Significant Patterns", *Machine Learning, 68 (1),* pp 1-33.

[97] Wickens, T. D. (1989) "Multiway Contingency Tables Analysis for the Social Sciences", Lawrence Erlbaum Associates.

[98] Wooldridge, M. (2000) "Intelligent Agents", *Multi-agent Systems: A Modern Approach to Distributed Artificial Intelligence,* Weiss, G. (ed.). The MIT Press, pp 27-77.

[99] Younis, H. L. S, and Littman, M. (2004) "PPDDL 1.0: The Language for the Probabilistic Part of IPC-4", *Proc. of the 4th International Planning Competition, (ICAPS 04),* pp 1-4.

[100] Zettlemoyer, L. S., Pasula, H. M. and Kaelbling, L. P. (2005) "Learning Planning Rules in Noisy Stochastic Worlds", *Proc. 20th National Conference on Artificial Intelligence (AAAI-05)*, pp 911-918.