Child, C. H. T. & Stathis, K. (2005). SMART (Stochastic Model Acquisition with ReinforcemenT) learning agents: A preliminary report. Lecture Notes in Computer Science: Adaptive Agents and Multi-Agent Systems II, 3394, pp. 73-87. doi: 10.1007/978-3-540-32274-0_5

SMART (**S**tochastic **M**odel **A**cquisition with **R**einforcemen**T)** Learning Agents: A Preliminary Report

Christopher Child* and Kostas Stathis
Department of Computing,
School of Informatics,
City University, London
{c.child,k.stathis}@city.ac.uk

Abstract

We present a framework for building agents that learn using SMART, a system that combines stochastic model acquisition with reinforcement learning to enable an agent to model its environment through experience and subsequently form action selection policies using the acquired model. We extend an existing algorithm for automatic creation of stochastic strips operators (Oates et. al 1995) as a preliminary method of environment modelling. We then define the process of generation of future states using these operators and an initial state and finally show the process by which the agent can use the generated states to form a policy with a standard reinforcement learning algorithm. The potential of SMART is exemplified using the well-known predator prey scenario. Results of applying SMART to this environment and directions for future work are discussed.

1 Introduction

Reinforcement learning has been shown to be a powerful tool in the automatic formation of action policies for agents (Kaelbling et. al. 1996). There are two main approaches: Value learning (V-Learning) which assigns a value to each state in the system and Q-Learning, which assigns a value to each state action pair (Sutton and Barto 1998). V-learning assigns state values by propagating back rewards received in future states after taking an *action* according to a *policy*. V-learning is limited in application because it requires a model of the world in order to predict which state will occur after each action. Q-Learning is more widely applicable because it assigns rewards to a state action pair. The agent is therefore not required to predict the future state and does not require a model. Often it is impossible for the designer of an agent to provide a model of the environment. Even if the environment has been designed in software, the transition from state to state may be impossible for the designer to predict due to factors such as the action of other agents.

In this paper we investigate "stochastic model acquisition". We define this to be any system which enables an agent to acquire a model of its environment from the environment. To be more precise we are modelling the agent's perception of the environment because this is all the information which it has access to. Initially the agent has knowledge of the actions it can perform, but not the effects, and has a percepive function that maps a world state to a set of percep variables. The agent's task is to discover which variables are effected by its actions, the conditions under which these effects will occur, and an associated probability. Using this model the agent can develop an action policy to achieve a goal using reinforcement learning. Similar work on modelling in deterministic environments has been called "discovery" (Shen 1993) and "constructivist AI" (Drescher 1991).

A well-studied model based reinforcement learning architecture is Dyna-Q (Sutton & Barto 1998). The algorithm suffers from the disadvantage that it relies on statistical measures of entire state transitions rather than state variable transitions. These require multiple visits to each state in a stochastic environment before an accurate model can be built. As the number of states increases exponentially with the number of state variables in the system, the method quickly becomes impractical.

Our approach is motivated by a number of observations:

1. V-learning reduces the state space for the reward function as compared to Q-learning by an order of magnitude;
2. Construction of a model allows us to make predictions about state action pairs that have previously not been visited;
3. Learning in the model, rather than through environment interaction can reduce the learning time of reinforcement learning algorithms.

* Corresponding author.

4. Changes in the agents goals (reward function) result in re-learning a state-value map from the model, rather than re-learning in the environment. A new policy can therefore be formed with no environment interaction.

Several methods have been evaluated for representing a stochastic environment using a factored state model (Boutilier et. al. 1999). The most compact representation is identified as the use of probabilistic STRIPS operators (Hanks 1990). Other methods they describe are generally based on two-tier Bayesian networks. These require exponentially large storage for the probability matrix as the number of state variables increases, unless the structure of variable dependencies is known. This structure can be learned but is a research area in itself. The MSDD algorithm (Oates et. al. 1995) has been shown to be an effective method of learning stochastic STRIPS operators and will therefore be used in this work.

Having generated a set of stochastic STRIPS operators, the next stage is to generate expected world states using these STRIPS operators. Finally we use a standard reinforcement learning algorithm to create a value map and therefore an action policy for the agent. This results in a fully functional agent mind, generated with no human intervention.

2 Motivating Example

The broad motivation for this research is towards automatic action selection mechanisms for robotics, software agent and computer game applications. For the purposes of this extended abstract we have selected the well known predator prey scenario, which is a very simple example of an agent which has a limited number of actions and a restricted perceptual system.

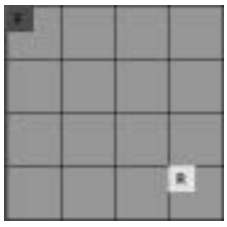

Figure 1 : Simple predator prey scenario. F indicates the predator (fox) and R the prey (rabbit).

We will be using a simple predator prey scenario (Figure 1). There is a four by four grid surrounded by a "wall". There is *one* predator and *one* prey. The predator will be assumed to have caught the prey when it lands on the same square. In this instance the prey will simply select a random action. Both predator and prey have four actions: move north, east, south and west. An action has the effect of moving the agent one square in the selected direction, unless there is a wall, in which instance there is no effect. The predator and prey move in alternate turns. The agent's percep gives the contents of the four squares around it and the square it is on. Each square can be in one of three states: empty, wall or agent. For example a predator agent which is situated in the north west corner of the grid with a

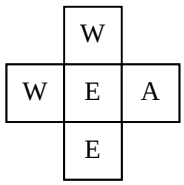

Figure 2: The predator agent percep

prey to the east would have the percep <WALL, AGENT, EMPTY, WALL, EMPTY> corresponding to the squares to the north, east, south, west and under respectively, as shown in figure 2.

3 Framework

The stages of the agent system are as follows: stochastic model acquisition, state generation and policy generation.

3.1 Stochastic Model Acquisition

3.1.1 Stochastic STRIPS operators

The STRIPS planning operator representation has, for each action, a set of preconditions, an "add" list, and a "delete" list (Fikes and Nilsson 1971). The STRIPS planner was designed for deterministic environments, with the assumption that actions taken in a state matching the operator's preconditions would consistently result in the state changes indicated by the operators add and delete lists. In a non-deterministic environment a less restrictive view is taken, allowing actions to be attempted in any state. The effects of the action then depend on the state in which it was taken and are influenced by some properties external to the agents perception which appear random from the agent's perspective.

We follow the format for stochastic STRIPS operators used by Oates & Cohen (Oates and Cohen 1996). A stochastic STRIPS operator takes the form:

$O = < a, c, e, p >$

where a specifies an action, c specifies a context, e the effects and p the probability. If the agent is in a state matching the context c, and takes the action a, then the agent will observe a state matching the effects e with probability p.

Contexts and effects of operators are specified as a list of tokens representing the percep and the action of the agent in the order described in section 2, with the addition of a wildcard symbol (*) denoting irrelevance, which matches any token.

As an example of the use of wildcards, consider the following operator:

<MOVE EAST, (* WALL * * *), (* WALL * * *)> *Prob:*1.0

In this example: a is MOVE EAST, c is (* WALL * * *), e is (* WALL * * *) and p is 1.0.

This operator specifies that if the agent chooses to move north and the contents of the square to the north is detected as WALL then the contents of the square to the north of the agent on the next time step will still be a wall, with probability 1.0. The wildcards specify that this condition is irrelevant of anything else that the agent observes[1]. The percep order is as specified in section 2.

3.2 Learning STRIPS operators

We have chosen to use Multi-Stream Dependency Detection (MSDD) (Oats et. al. 1995) to learn the STRIPS operators in this context. Although inductive logic programming (ILP) (Muggleton 2000) is powerful in its domain area of learning predicate logic rules, it is still an open research area to generate reliable stochastic logic rules with the system. MACCENT is another inductive logic system specifically designed for learning stochastic rules (Dehaspe 1997), but is not well suited to large rule sets. MSDD has been chosen for its suitability to deal with the domain area, and has previously been shown to be able to generate stochastic STRIPS operators from data (Oates and Cohen 1996).

3.2.1 MSDD

Formal statements of both the MSDD algorithm and its node expansion routine are given in the algorithms below. MSDD is a batch algorithm and uses H, the precep data observed by the agent in the preconditions-effects format shown in section 3.1.1. The function f evaluates the best node to expand next and typically counts the co-occurrence of the node's preconditions and effects. This requires a complete pass over the data set H.

```
MSDD (H, 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(H,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 ∈ Ti 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
```

This algorithm does not specify which children should be generated before others, but does ensure that each dependency is explored only once, and facilitates pruning of the search. For example, all descendants of the node

<*, (WALL * * * *), (WALL * * * *)>

can be pruned because there is no need to explore rules with a wildcard in the action position, and all descendants of this node will be expanded from the rightmost wildcard resulting in children with no action.

We have made two changes to standard MSDD. The first is in EXPAND (b.iii). We check that the generated child matches at least one observation in the database before adding it to children. For example, MSDD can generate the rule <*, (WALL WALL WALL * *), (* * * * *)>, but in our environment the agent can only observe a maximum of two walls (when it is positioned in the corner of the map). 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 it's children as a consequence. This prevents the generation of a large number of incorrect rules, which would have been eliminated in the "filter" stage of MSDD.

The second change is that the effect part of the rule is only allowed to have one non-wildcard element. We have made this change because a very large number of rules are generated by standard MSDD. Combining individual effect fluents can generate complete successor states. This has the disadvantage that illegal states can be created, such as <WALL WALL WALL * *>. These can, however, be eliminated using constraints (section 3.3).

3.2.2 Filter

The second stage of MSDD is the "filter" process, which removes specific rules already covered by more general ones. For example, <C_1 C_2 * * *> is a more specific version of <C_1 * * * *>. If the condition C_2 has no significant effect on the probability of the rule then it is unnecessary. For example, MSDD could generate two rules as follows:

<MOVE NORTH, (WALL WALL * * *), (* WALL * * *)> *Prob:* 1.0
<MOVE NORTH, (* WALL * * *), (* WALL * * *)> *Prob:* 1.0

Both of these rules tell us that, if the agent moves north and there was a wall to the east, it will observe a wall to the east on the next move. The extra information that there was a wall to the north does not affect the agent's subsequent observation. More general operators are preferred because a reduced number of rules can cover the same information. For operators which do not have a probability of 1 we are testing, for an operator $O = <a, c, e, p>$, whether $Prob(e \mid c_1, c_2, a)$ and $Prob(e \mid c_1, a)$ are significantly different. If not, the general operator is kept the specific one discarded.

[1] Note that the agent's percep does not include the agent itself.

```
FILTER (D, H, g)
1. sort D in non-increasing order of generality
2. S = {}
3. while NOT_EMPTY(D)
   a. s = POP(D)
   b. PUSH (s, S)
   c. for d ∈ D do
      if SUBSUMES(s, d) and G(s, d, H) < g then
         remove d from D
4. Return S
```

Where D is the set of dependency operators generated by MSDD. H is the history of observations made by the agent. SUBSUMES(d_1, d_2) is a Boolean function defined to return true if dependency operator d_1 is a generalisation of d_2. G(d_1, d_2, H) returns the G statistic to determine whether the conditional probability of d_1's effects given its conditions is significantly different from d_2's effects given its conditions. The parameter g is used as a threshold, which the G statistic must exceed before d_1 and d_2 are considered different[2].

For example, dependency d_1 below returns a very low G statistic when compared with d_2:

d_1:<MOVE NORTH, (WALL * EMPTY * AGENT), (* * * * AGENT) >
d_2:<MOVE NORTH, (WALL * * * AGENT), (* * * * AGENT) >

The additional condition that there is an EMPTY square to the south does not effect the probability of an agent being present on the square it inhabits after a move north (which will be the same square because there is a wall to the north). The predator agent already has the information that the prey is in the same square and that there is a WALL to the north, so the square to the south will either be the prey or be empty irrelevant of the empty square to the south. For an explanation of calculation of the G statistic see Oates et. al. (1995).

3.2.3 Rule Complements

When running MSDD on data collected from the predator prey scenario, we observed that the "filter" process occasionally filters rules and not their complements. For example the filter process could filter d_2 below, but leave d_1:

d_1: <MOVE NORTH, (WALL WALL * * AGENT), (* * * * AGENT) >
d_2: <MOVE NORTH, (WALL WALL * * AGENT), (* * * * EMPTY) >

This would cause a problem in the state generation (section 3.3), because the rule generates only states with agents present. To avoid this we have added a further step to the algorithm in order to search through all generated rules checking that their complement rules are present. This process is less problematic in our system because rules only have effects in one fluent.

[2] A value of 3.84 for g tests for statistical significance at the 5% level.

```
AddRuleComplements(D, H)
for d ∈ D do
  f = d.effect
  for fValue ∈ possibleValues(f)
  if fValue not equal to f.value
    newRule = copy of d with d.effect set to
              fValue
    if newRule is not present in D
       if newRule has match in H
          add newRule to D
```

The above algorithm goes through all rules in the learned dependencies, D, checking that all possible values of its effect fluent are either present in D already or do not match any observations in the history H. If a missing rule is found it is added to D.

3.3 Generating States from learned rules

The state generator function of the SMART learning process generates all possible next states, with associated probabilities, for an action that the agent could take in a given state. These states are generated using the rules learned by MSDD from the history of observations. This state generation process is necessary for the agent to generate a state value map using reinforcement learning (section 3.3.5).

Our modified implementation of the MSDD algorithm generates a set of rules with only one fluent in the effects part in order to reduce substantially the number of rules that must be evaluated. When we match these against some initial conditions, such as:

<MOVE EAST, (EMPTY WALL WALL EMPTY EMPTY)>

The subset of the generated rules matching these conditions is shown in *Table 1*.

Table 1: Subset of the generated dependency rules matching the condition <MOVE EAST, (EMPTY WALL WALL EMPTY EMPTY)>. E = EMPTY, W = WALL and A = AGENT. The table shows the rule set after removing general rules (section 3.3.1).

Conditions						***Effects***					***Pr***
Action	**N**	**E**	**S**	**W**	**U**	**N**	**E**	**S**	**W**	**U**	
EAST	*	W	*	*	*	*	W	*	*	*	1.0
EAST	*	*	W	*	*	*	*	W	*	*	1.0
EAST	E	W	*	E	*	*	*	*	A	*	0.07
EAST	E	W	*	E	*	*	*	*	E	*	0.93
EAST	E	W	*	*	E	A	*	*	*	*	0.03
EAST	E	W	*	*	E	E	*	*	*	*	0.97
EAST	*E*	*	*W*	*E*	*	*A*	*	*	*	*	*0.06*
EAST	*E*	*	*W*	*E*	*	*E*	*	*	*	*	*0.94*
EAST	E	W	*	E	E	*	*	*	*	E	1.0

Notice that the rule set correctly tells us that moving east when there is a wall to the east results in us still observing a wall to the east. Also moving east with a wall to the south results in a wall to the south irrespective of any other contextual information. The rule set captures all the information we require for the given state and action and has the advantage that rules are also applicable to many other states.

The generated rule set for a given situation can provide several rules for each output fluent. To generate individual states from these rules we have to decide which rules are more relevant in the given situation. The following sections describe the process by which we choose rules, and why the decision was made to process the rules in this way. The stages of the state generator are:

1. Remove general rules covered by specific.
2. Remove rules with less specific effects.
3. Generate possible states and probabilities.
4. Remove impossible states using constraints and normalise the state probabilities.

The output of the state generator is a table of states and associated probabilities.

3.3.1 Remove General Rules Covered by Specific

The subset of the generated rules that apply to a specific state action pair may contain rules with different conditions applicable to the same effect fluent. If the environment has not been well explored by the agent, a general rule may be more reliable because a specific rule will not have been encountered as often in the history of observations made by the agent. If, however, the agent has been allowed to explore the environment extensively, as was the case in our experiments, we can assume that a specific rule is likely to contain more relevant information than a general one. An expansion to the system would be to choose less specific rules depending on the statistical confidence measure of each rule. This is a subject for further research.

General rules are removed by searching through the rest of the dependencies for other nodes with an effect for the same fluent. If another rule is found which is more specific the general rule is removed. Rules with equal specificity are not removed.

```
RemoveRulesCoveredBySpecific(rules)
sort rules in non-increasing order of generality
for rule ∈ rules
  for testRule ∈ rules after rule
    if effectFluent of rule is same
         position as effectFluent of testRule
      if rule.wildcards > testRule.wildcards
    remove rule from rules
```

Table 2 gives an example of rules that were removed to give the rule set in Table 1. The rules were removed because a more specific rule was available for the same effect fluent.

Table 2: The first two rules in the above table were removed because the specific rule (in bold) covered them.

Conditions						***Effects***					***Pr***
Action	**N**	**E**	**S**	**W**	**U**	**N**	**E**	**S**	**W**	**U**	
EAST	*	*	*	E	*	*	*	*	*	A	0.04
EAST	*	*	*	E	*	*	*	*	*	E	0.96
EAST	**E**	**W**	*****	**E**	**E**	*****	*****	*****	*****	**E**	**1.0**

3.3.2 Remove Rules With Less Specific Effects

MSDD generates rules with equally specific conditions but less specific results. Rules which are more specific in the effects part are providing us with the information that the given effect cannot arise in this context. We can therefore eliminate the less specific effects. The rules shown in italics in Table 1 are removed by this part of the state generation process, because other rules exist with equal specificity of effect.

```
removeLessSpecificOutcome(rules)
for rule ∈ rules do
  effectP = position of effect in testRule
  NumEffects = countRules(rule.conditions,
                          effectP)
  for other ∈ matchingNodes do
     if other has same effectP and same
               conditions
        otherNumEffects
            = countRules(other.conditions,
                         effectP)
     if otherNumEffects >= NumEffects
        removeRules(other.conditions, effectP)
      else if otherNumbOfEffects > 0
         removeRules(rule.conditions, effectP)
```

If we do not remove rules with less specific effect in this way the state generator can produce states which could not occur in the real environment. Consider for example the rules in Table 3.

Table 3: Removing rules with less specific outcomes. Rules in Italics are removed because their outcome is less specific than the other rules with the same effect fluent.

Conditions						***Effects***					***Pr***
Action	**N**	**E**	**S**	**W**	**U**	**N**	**E**	**S**	**W**	**U**	
NOR	E	E	W	*	E	E	*	*	*	*	0.94
NOR	E	E	W	*	E	A	*	*	*	*	0.06
NOR	*E*	*E*	*	*E*	*E*	*E*	*	*	*	*	*0.07*
NOR	*E*	*E*	*	*E*	*E*	*A*	*	*	*	*	*0.05*
NOR	*E*	*E*	*	*E*	*E*	*W*	*	*	*	*	*0.88*

The rules removed by this process, shown in italics, state that a move north could result in a wall to the north. In fact there is a wall to the south in the original state. A move north could, therefore, only result in an empty square or an agent, as there is no situation in which the agent can move

from one wall to another opposite in a single move in the environment.

3.3.3 Generate Possible States

The possible states are generated by:

1. Creating a new state from each combination of effect fluent values in the remaining rules.
2. Multiply the probability of each effect rule to generate the probability of each state.

The states generated from the rules in Table 1 are as follows:

Table 4: Possible after states for the given state and action generated from the rules in Table 1.

NORTH	EAST	SOUTH	WEST	UNDER	Pr:
EMPTY	WALL	WALL	EMPTY	EMPTY	~0.91
EMPTY	WALL	WALL	AGENT	EMPTY	~0.07
AGENT	WALL	WALL	EMPTY	EMPTY	~0.02
AGENT	*WALL*	*WALL*	*AGENT*	*EMPTY*	*~0.001*

There were two rules for the "north" fluent with results EMPTY and AGENT, and two rules for the "west" fluent with results EMPTY and AGENT. The other rules had one result each resulting in a total of: 2 * 1 * 1* 2 * 1 = 4 possible states. The probabilities are found by multiplying together the probabilities of the rules which resulted in these fluents.

3.3.4 Remove Impossible States with Constraints

Some of the states generated could not occur in the domain area. For example in the predator prey scenario the operators may generate a percep with two agents, when there is only one agent in the world. We would ultimately like our agent to generate its own constraints that tell it which world states are impossible. A rule such as IMPOSSIBLE (* * AGENT AGENT *) would allow elimination of the impossible world states generated. For the purposes of this paper, we will be using a user-defined set of constraints. If we do not use these constraints the erroneous generated states can propagate to create world states where there are five prey agents, or walls surround the predator agent, and the model becomes meaningless as it is too far detached from the real world states. Currently our system simply removes impossible states by checking that each generated state does not contain more than one agent, or walls opposite each other. This method breaks the principle of an autonomous learning agent that learns a model of its environment without human intervention. A constraint generator will therefore be the subject of future research.

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. The state in italics in Table 4 is removed by this process because it contains two agents.

3.3.5 Removing Unused Rules

In the present system the state generator has to continually search through the dependency list and remove general rules which are covered by specific ones. We could reduce both the size of the rule set, and the time taken to generate states by removing general rules which are never used to generate states from the rule set as follows:

Starting from the most general rule, in order of increasing specificity, we check to see if there is a set of more specific rules entirely covering the possible observed values of each wildcard fluent. If, for example, the rule set below exists:

d_1: <MOVE NORTH, (* * * * AGENT), (* * * * AGENT) >
d_2: <MOVE NORTH, (* * * * AGENT), (* * * * EMPTY) >

The following more specific rule set would cover the above rules and cause them to be unused:

d_1: <MOVE NORTH, (WALL * * * AGENT), (* * * * AGENT) >
d_2: <MOVE NORTH, (WALL * * * AGENT), (* * * * EMPTY) >
d_3: <MOVE NORTH, (EMPTY * * * AGENT), (* * * * AGENT) >
d_4: <MOVE NORTH, (EMPTY * * * AGENT), (* * * * EMPTY) >

Notice that the more specific rule set covers WALL and EMPTY, but does not have an AGENT value. In our predator prey scenario it is not possible to have two agents in one percep. These rules therefore cover all fluent values with the remaining value finding no matches in the database. This feature has not been implemented in the current system.

3.4 Generating Policies

We use standard reinforcement learning techniques to generate a policy for the agent. The acquired model allows us to use value iteration, which is a simple and efficient method of generating a value map for the agent (Sutton & Barto 1998). The update equation for value iteration is given by:

$$V_{k+1}(s) = \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$$

The value of state s on pass k +1 of the value iteration is calculated by taking the maximum valued action. The value of the action is equal to the sum for s' of the probability of action leading from state s to s' multiplied by the reward plus the discounted value of state s' on pass k.. In order to generate a value map, we start with a state generated by a random initial position of the predator prey scenario and add this to the value map. A single entry in the value map is as follows:

State, Value, Reward

State is a percep (e.g. <WALL, WALL, EMPTY, EMPTY, EMPTY>). *Reward* is set to a positive value if there is an AGENT in the last position of the percep (corresponding to the predator catching the prey) and zero or a negative value otherwise. *Value*, for each state is then generated by repeated application of the following algorithm, with the next state to refine picked at random from those generated in the getCreateStateValue step of the algorithm below.

```
refineValue(state, actions, valueMap)
maxUtility = 0
for action ∈ actions do
  nextStatesAndProb = generateStates(state,
                                     action)
  actionUtility = 0
  for afterState ∈ nextStatesAndProb
    getCreateStateValue(afterState)
    actionUtility += afterState.probability *
       (afterState.value * γ + afterState.reward)
    if (actionUtility > maxUtility)
      maxUtility = actionUtility
setStateValue(valueMap, state, maxUtility)
```

4 Empirical Results

Performance of the policy generated by SMART learning was assessed by tested against a standard Dyna-Q algorithm (Sutton and Barto 1998). Dyna-Q is a model based learning system, which uses a state map generated by recording the frequency with which each action in each state leads to the next state. Dyna-Q has previously been applied to the predator prey scenario presented in this paper (Varsy 2002). We therefore repeat the test conditions used for this work using the SMART framework, for the purposes of comparison.

4.1 Test Conditions

An experimental run consisted of a sequence of trials or episodes that end after the prey has been captured. In the first trial of each run, the predator and prey are given the start position as indicated by Figure 1. The result of an experiment was the number of steps required for the predator to catch the prey averaged over 30 runs. Each run was 2000 steps, giving 1000 moves for both the predator and prey agents.

In our test environment we allowed the agent to gather a perceptual history (*H*) from 10,000 iterations of the environment. Our extended MSDD algorithm was subsequently run on the observation history to generate stochastic STRIPS operators. The state generator and reinforcement learning algorithm were then applied for 2,000 iterations to generate a value map. *Reward* was set to 1.0 for a state where the prey agent is "under" the predator, and –0.1 otherwise. The discount factor (γ) was 0.9. The environment was then run according to the test conditions. On each environment update the predator agent picked the highest utility action. This was achieved using a similar process to the selection of highest utility action in the "refineValue" algorithm (section 3.4).

Table 5: Average life-span of prey under test conditions

Action Method	**Prey life-span**
Random moves	16.37
Dyna-Q	7.04
SMART learning	9.72

Both predator and prey taking random moves resulted in the predator being on the same square as the prey every 16.37 moves. This result is expected. There are 16 squares in the grid and the predator will randomly occupy the same square roughly once every 16 moves. The extra fraction is probably due to the start positions being at opposite ends of the grid.

Using Dyna-Q the predator caught the prey in an average of 7.04 moves. Dyna-Q was required to learn a policy during the run, and, as the predator and prey take alternate turns, half the moves are taken by the prey and although random, are likely to result in the prey evading the predator.

SMART learning resulted in the predator catching the prey in an average of 9.72 moves. Initially this might appear to be a low score when compared to Dyna-Q, because the agent enters the world with a fully formed policy. An ideal state action policy should result in the predator being able to move onto the prey's square every time it moves once it has caught it the initial time. This would result in a prey life-span of approximately 2 moves. The SMART learner, however, is a state-value method, and does not have access to the immediate effect of its action as it perceives just

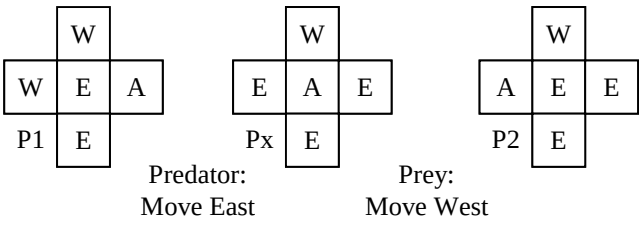

Figure 3: Order of perceptions in predator-prey scenario

before it moves and not just after. The prey takes a move between these two perceptions, so the predator is only able to learn by an approximate method.

Figure 3 shows order of perceptions from the perspective of the predator. The predator initially has the perception P1. The move east action takes the predator onto the agent's square. The predator, however, does not observe the percep Px, because it is the prey's turn to move. When the predator observes again at P2 the prey has moved out of the capture square. Methods such as Dyna-Q suffer less from this problem, because reward is associated with the state, action pair (e.g. <P1, Move East>) and the predator

is therefore able to associate rewards with the actions which produced them.

Despite this problem, SMART learning enabled the agent to form a good policy in the predator prey environment and the failure to form an optimum policy owes more to the experimental conditions than a problem with the learning technique itself.

5 Conclusions and Future Work

We have presented a framework for stochastic model acquisition, which promises to be a powerful extension to the reinforcement learning paradigm. We have shown how to overcome some of the problems encountered when attempting to generate next states from automatically acquired STRIPS operators and demonstrated that action policies can be developed using a model represented by these operators.

The ability to learn a model of the environment through experience automates an otherwise difficult or impossible process for an agent designer. Future experiments aim to demonstrate that the agent can keep important experience when its goals are changed and the designer is able to change the reward structure and learn a new policy without the need for expensive interaction with the environment.

The use of a rule learning method to acquire a model provides an accessible format for a human designer. If the system is not performing as the designer wishes the rules can be investigated and anomalies spotted more easily than with a black box learning system such as a neural network.

Subjects of further research will include:

- Testing the system in an environment more suited to model based learning, in which the results of action are immediately perceivable by the agent.
- Evaluation of rule learning methods in environments with greater independence between state variables, where stochastic STRIPS operators are likely to more efficiently compress the environment model.
- The addition of a parameterised value learning system to estimate state values, allowing compression of the state value map (Tesauro 1994)
- Investigation of stochastic predicate logic rule learning methods to learn stochastic situation calculus rules (Muggleton 2000). This would also require the use of a relational reinforcement learning method (Dzeroski et. al. 2000) to learn state values.

6 Acknowledgements

Chris Child would like to acknowledge the support of EPSRC, grant number 00318484.

References

Boutilier, C. Dean, T.Hanks, S. 1999. Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research* 11: 1-94.

Dzeroski, S. and De Raedt, L. and Blockeel, H. 1998. Relational Reinforcement Learning. *International Workshop on Inductive Logic Programming*.

Drescher, G.L. 1991. *Made-Up Minds, A Constructivist Approach to Artificial Intelligence*. The MIT Press.

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): 189-208.

Hanks, S. 1990. Projecting plans for uncertain worlds. Ph.D. thesis, Yale University, Department of Computer Science.

Kaelbling, L. P. and Littman, H.L. and Moore, A.P. 1996. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research* 4: 237-285.

Muggleton, S.H. 2000. Learning Stochastic Logic Programs. *Proceedings of the AAAI2000 Workshop on Learning Statistical Models from Relational Data*, L. Getoor and D. Jensen, AAAI

Oates T., Schmill, M.D., Gregory, D.E. and Cohen P.R. 1995. Detecting complex dependencies in categorical data. Chap. in *Finding Structure in Data: Artificial Intelligence and Statistics V*. Springer Verlag.

Oates, T. and Cohen, P. R. 1996. Learning Planning Operators with Conditional and Probabilistic Effects. *AAAI-96 Spring Symposium on Planning with Incomplete Information for Robot Problems*, AAAI.

Sutton, R.S., and A.G. Barto. 1998. *Reinforcement Learning: An Introduction*. A Bradford Book, MIT Press.

Shen, W. 1993. Discovery as Autonomous Learning from the Environment. *Machine Learning* 12: 143-165.

Tesauro, G.J. 1994. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation* 6, 2: 215-219.

Varsy, R. 2002. Extending Planning and Learning Through Reinterpretation of World Model. M.Sc. thesis, City Univesity.