The Apriori Stochastic Dependency Detection (ASDD) Algorithm for Learning Stochastic Logic Rules.

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

Abstract. *Apriori Stochastic Dependency Detection* (ASDD) is an algorithm for fast induction of stochastic logic rules from a database of observations made by an agent situated in an environment. ASDD is based on features of the Apriori algorithm for mining association rules in large databases of sales transactions [1] and the MSDD algorithm for discovering stochastic dependencies in multiple streams of data [15]. Once these rules have been acquired the *Precedence* algorithm assigns operator precedence when two or more rules matching the input data are applicable to the same output variable. These algorithms currently learn propositional rules, with future extensions aimed towards learning first-order models. We show that stochastic rules produced by this algorithm are capable of reproducing an accurate world model in a simple predator-prey environment.

1 Introduction

This paper introduces the *Apriori Stochastic Dependency Detection* (ASDD) algorithm for fast induction of stochastic logic rules from a database of observations. The focus of our research is on methods by which a logic-based agent can automatically acquire a rule-based model of a stochastic environment in which it is situated from observations and use the acquired model to form plans using decision theoretic methods. Examples in this paper are geared towards this research, but the algorithm is applicable to induction of stochastic logic rules in the general case.

The key feature of this algorithm is that it can eliminate candidate n element rules by reference to n-1 element rules that have already been discounted without the need to for expensive scans of the data set. This is achieved via a breadth first search. Rules are discounted at each level of the search if they do not occur regularly in the data set or the addition of extra constraints has no statistical significance on their performance.

Although research in stochastic rule induction is in its infancy, some previous research includes MSDD [14], ILP [12], and the schema mechanism [2]. For a discussion on the topic see [10].

* Corresponding author.

Our research is motivated by the observation that rule based methods in decision theoretic planning, such as stochastic STRIPS operators (SOPs) promise to be a highly efficient method of representing a world model for an agent in a stochastic environment [2]. The main advantage of SOPs is that they provide a solution to the frame problem which other methods in this area do not address [11].

Previous research in automatic acquisition of stochastic environment models has been focused on either explicit state space models or dynamic Bayesian networks (DBNs). State space models record the relative frequency with which each action available to an agent leads to a next state from an initial state [16]. These methods do not scale up well because each environment state must be explicitly enumerated. Dynamic Bayesian networks are an example of a factored state approach, in which the state space is modeled as a set of nodes representing state variables, and dependencies represented as connections. Although methods exist for modelling DBNs from data [13] the representation must explicitly assert that variables unaffected by an action persist in value and therefore suffers from the frame problem. Variables which are unaffected by an action in the probabilistic STRIPS representation, however, need not be mentioned in the actions description [2].

In order to give context to the ASDD algorithm, an example predator-prey domain and probabilistic strips operators (PSOs) are first introduced, which will form the basis of examples in the remainder of the paper. Section 2 describes the ASDD algorithm for stochastic rule induction. Section 3 describes the Precedence algorithm for operator precedence. Section 4 describes the process of state generation from PSO operators. Section 5 gives results comparing the algorithms performance against MSDD and a state space method. Conclusions and future work are presented in section 6.

1.1 Example Predator Prey Domain

The environment consists of 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. The prey selects a random action at each move. 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 simultaneous turns. The agent's percept gives the contents of the four squares around it and the square under it. Each square can be in one of three states: empty, wall or agent. For example a predator agent has a wall to the west and a prey to the east would have the percept {EMPTY_NORTH, AGENT_EAST, EMPTY_SOUTH, WALL_WEST, EMPTY_UNDER} corresponding to the squares to the north, east, south, west and under respectively (shown in figure 1).

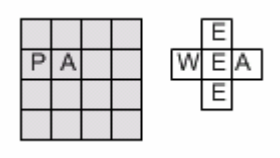

Fig. 1. Simple predator prey scenario. Predator and prey in a 4 by 4 grid. P indicates the predator and A the prey agent. The percept from the predator's perspective is shown to the right.

1.2 Probabilistic 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) [6]. 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 operator's 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.

The following format for a stochastic STRIPS operator is an adaptation of that used by Oates & Cohen [15] to the form of stochastic logic programs (section 1.3). A stochastic STRIPS operator takes the form: *prob: e* ← *a, c,* where *a* specifies an action, *c* specifies a context, *e* the effects and *prob* the probability of the effects. 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 *prob.*

The agent is assumed to have a set of *n* possible actions, $A = \{a_1, \ldots, a_n\}$ and can perceive *m* possible state variables $P = \{p_1, \ldots p_m\}$, each of which can take on a finite set of possible values. Let $p_i = \{p_{i1}, \ldots, p_{ik}\}$ be the values associated with the i^{th} variable. The context, *c*, of an operator is specified as a set of variables from P representing the perception of the agent. In order to restrict the number of possible operators, *e* is defined to be a *single* variable for each operator, again taken from the set P. A combination of single variable operators is, however, sufficient to generate a full percept.

In the predator prey domain:

- A = {MOVE_NORTH, MOVE_EAST, MOVE_SOUTH, MOVE_WEST}
- P = {NORTH, EAST, SOUTH, WEST, UNDER}
- P_{NORTH} = {EMPTY_NORTH, WALL_NORTH, AGENT_NORTH}
- P_{EAST}, P_{SOUTH}, P_{WEST}, P_{UNDER} follow the same form as P_{NORTH}

1.3 Stochastic Logic Programs

Stochastic logic programs (SLPs) are first-order logic program extensions of stochastic grammars. Although ASDD is currently not able to learn first-order programs,

the full SLP representation is presented, with the eventual goal of this research being to learn programs of this nature. Muggleton [12] defines the syntax of an SLP in as follows:

"An SLP, *S*, is a set of labelled clauses *p:C* where *p* is a probability (i.e. a number in the range [0,1] and C is a first-order range-restricted clause. The subset S_p is of clauses in *S* with predicate symbol *p* in the head is called the definition of *p*. For each definition S_p the sum of probability labels π_p must be at most 1. *S* is said to be complete if $\pi_p = 1$ for each *p* and incomplete otherwise. *P(S)* represents the definite program consisting of all the clauses in S, with labels removed."

2 Apriori Stochastic Dependency Detection (ASDD)

ASDD is based on the Apriori algorithm for mining association rules (section 2.1), and the MSDD algorithm for finding dependencies in multiple streams of data. MSDD has previously been applied to the problem of learning probabilistic STRIPS operators in [15] [4].

2.1 The Apriori method for Association Rule Mining

The Apriori algorithm was designed to address the problem of discovering association rules between items in a large database of sales transactions. A record in these databases typically consists of a transaction date and the items bought in the transaction (referred to as *basket* data). An example of association rule is that 98% of customers purchasing tyres and auto accessories also purchase automotive services [1]. The form of this rule is similar to a stochastic logic rule of the form: 0.98: Automotive Services ← Tyres, Accessories. The Apriori algorithm and its descendants have been shown to scale up to large databases and methods also exist for incrementally updating the learned rules [7][3]. For a survey see [6]. These features are highly desirable to probabilistic STRIPS learning with its need to process a large database of perceptions, and incrementally improve these rules as the agent receives new data. The language used to describe ASDD (2.2) has been chosen to reflect that used in [1]. The algorithm is largely similar, but has an additional *aprioriFilter* step (2.2.5) 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.

2.2 Apriori Stochastic Dependency Detection (ASDD)

The task of learning probabilistic STRIPS operators proceeds as follow: The sets *P* and *A* are as defined in section 1.2. Let *D* be a set of perceptual data items (*PDIs*) from an agent, where each *PDI* is a triplet of the form $\{P_{t-1}, A_{t-1}, P_t\}$ i.e. the percept and action at time t-1 and the percept at time t. The elements of $P \cup A$ are collectively defined as *rule elements*.

A *PDI* contains rule element set X, if $X \subseteq P_{t-1} \cup A_{t-1}$. A *rule* is an implication from a rule element set to an effect, e, of the form $e \leftarrow X$, where $e \subseteq P_t$. In logic programing terms e is the *head* of the rule and X is the *body*.

A *PSO (prob:* $e \leftarrow X$) is a *rule* with an associated probability (*prob*). A *rule* holds in the data set D with *probability prob* if *prob%* of *PDIs* which contain X also contain e.

The *rule* $e \leftarrow X$, has *support* s in the perceptual data set D if s of PDIs in D contain $e \cup X$. *minsup* defines the minimum *support* a PSO has to display before it is admissible to the rule base[1].

The problem of discovering a PSO set can be separated into three sub-problems:

1. *Discover large rule element sets* at level k exhibiting *support* above *minsup*. The *support* for a rule element set is the number of *PDIs* that contain the rule element set. The level of a rule element set is defined as the number of rule elements it contains (section 2.2.1).
2. Combine rule element sets at level k to form a list of candidate sets for level $k+1$ using *aprioriGen*, which removes all candidates that cannot have minimum support (section 2.2.3).
3. After level 3, apply the *AprioriFilter* function to remove stochastic planning operators (rule element sets with a result element) at level k, which are *covered* by an operator at level $k-3$ (section 2.2.5).
4. Finally, *filter* the remaining rules to remove stochastic planning operators which are *covered* by a rule at any level (section 2.2.4).

2.2.1 Discovering Large Rule Element Sets

Discovering large rule element sets involves making multiple passes over the perceptual data set D. In the first pass (giving level k = 1) the support of individual rule element sets is counted to determine which of them are *large*, i.e. have minimum support. In each subsequent pass, large rule element sets from the previous pass (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 effect element (rule head) can be *filtered* by rules at the k-1th level to see if additional conditions have a significant effect on its probability (section 2.2.5). 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 considering only the rule element sets found to be large in the previous pass. Candidates with k rule elements are generated by rule element sets at the k-1 level. Any generated candidates containing a k-1 set which does not have minimum support are then removed in a the prune step, because any subset of a large set must also be large. This avoids the need for an expensive pass over the data set when generating candidates.

The notation is used in the following algorithms is:

---

[1] This definition of support is slightly different from the Apriori algorithm, in which support is defined as a percentage of the data.

– L[k]: Set of large k-rule element sets (those with minimum support). Each member of this set has three fields:
1. Elements: a set of rule elements
2. Support: the number of times the rule matched the database (if the set of rule elements has an effect (rule head).
3. BodySupp: the number of times the body of the rule matched the database
– C[k]: Set of candidate k-rule element sets (potentially large sets). Fields are identical to L[k].

2.2.2 The ASDD algorithm

The first part of the Apriori algorithm simply counts occurrences of single rule elements to determine large 1 rule element sets. A subsequent pass consists of the following 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 2.2.3).
2. The *support* of candidates in C[k] is counted by a database scan using the *subset* function, which returns the subset of C[k] contained in a PDI[2].
3. Rule element sets with below minimum support are removed.
4. If rule element set has no effect (head) bodySupp = support.
5. Rules, which are rule element sets with an effect element (rule head), are filtered against rules that subsume them at the level k-3 by the *aprioriFilter* function (section 2.2.5).

Finally, rules are filtered with a greater test for statistical significance by the *filter* function (section 2.2.4).

```
ASDD(D)
L[1] = {large 1-literalsets};
for (k = 2; L[k-1] ≠ ∅; k++) {
   Ck = AprioriGen(L[k-1]);                    //(1)
   for (pdi ∈ D) {                             //(2)
      Ct = Subset(Ck, pdi)
      for (c ∈ Ct)
         c.support ++;
   }
   L[k] = {c ∈ Ck | c.support ≥ minsup}        //(3)
   for (l ∈ L[k] where not HasEffect(l))
         l.bodySupp = l.support;
   if (k > 3)
      Ck = AprioriFilter(Ck, L[k-3]);          //(4)
}
ruleSet = ∪ for k of L[k];
return filter(ruleSet);
```

2.2.3 AprioriGen

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

There are two main steps:

[2] An efficient subset function is described in the original algorithm but is not used in the implementation tested here.

1. The join step joins L[k-1] with L[k-1] to form candidate rule sets C[k].
2. The prune step deletes generated candidates for which some (k-1) subset is not present in L[k-1].

For the purposes of rule generation, the following steps have been added:

1. Rules (rule element sets with an effect element) will have a body that is equal to one of the rules used to form them. In this case the bodySupp variable is copied to restrict the number of database passes required.
2. Effects are restricted to just one variable. If both L[k-1] rules have an effect variable (rule head) they are not combined (the *HasEffect* function will return true).

```
Join(L[k-1])
C[k] = ∅
for (p ∈ L[k-1]) {
   for (q ∈ L[k-1]) {
      generate = true;
      if (p == q) next q;
      if (HasEffect(p) and HasEffect(q)) next q;
      if (p.lastElement > last(q.elements)) {
         generate = false; next q; }
      for (i from 0 to num elements in p-2) {
         if (p.elements[i] ≠ q.elements[i])
            generate = false; next q;
      }
      if (!generate)
         next q;
      newC.elements = p.elements + last(q.elements);
      if (HasEffect(newC) {
         if (body(newC) == body(p)) newC.bodySupp = p.bodySupp;
         if (body(newC) == body(q)) newC.bodySupp = q.bodySupp;
      }
      add(C[k], newCandidate);
}}
return C[k]

Prune(Ck, L[k-1])
for (c ∈ C[k]) {
   forall (k-1 size subsets s of c) {
      if (s ∉ L[k-1]) delete c from C[k]
```

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

Example: L[3] rule element sets are (← indicates an effect element):

1. {MOVE_NORTH, AGENT_NORTH←, EMPTY_EAST},
2. {MOVE_NORTH, AGENT_NORTH←, WALL_SOUTH},
3. {MOVE_NORTH, EMPTY_EAST, WALL_SOUTH},
4. {MOVE_NORTH, EMPTY_EAST, WALL_NORTH},
5. {AGENT_NORTH←, EMPTY_EAST, WALL_SOUTH}.

The join step creates the C[4] rule element sets as follows: From a combination of 1 and 2: {MOVE_NORTH, AGENT_NORTH←, EMPTY_EAST, WALL_SOUTH). From a combination of 3 and 4: {MOVE_NORTH, EMPTY_EAST, WALL_SOUTH, WALL_NORTH}.

The prune step will delete the rule element set {MOVE_NORTH, EMPTY_EAST, WALL_SOUTH, WALL_NORTH} because the subset {MOVE_NORTH, WALL_SOUTH, WALL_NORTH} is not contained in L[3]. In the full data set this behaviour is observed

because the agent cannot perceive the conditions WALL_SOUTH and WALL_NORTH simultaneously. The algorithm is able to draw this conclusion without a further pass through the data.

2.2.4 Filter

The *filter* function was presented in (Oates and Coen) [15] as an extension to the MSDD algorithm. It removes rules that are *covered* and *subsumed* by more general ones. For example, the rule {Prob 1.0: WALL_NORTH ← MOVE_NORTH, WALL_NORTH, WALL_EAST} is a more specific version of {Prob 1.0: WALL_NORTH ← MOVE_NORTH, WALL_NORTH,} and therefore *subsumes* it. 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 WALL_EAST has no significant effect.

More general operators are preferred because they are more likely to apply to rules outside the original data set and a reduced number of rules can cover the same information. The test determines whether *Prob* (e| c_1, c_2, a) and *Prob* (e | c_1, a) are significantly different. If not, the general operator is kept the specific one discarded.

```
Filter (R, D, g)
sort R in non-increasing order of generality
S = {}
while NotEmpty(R)
   s = Pop(R)
   Push (s, S)
   for (r ∈ R)
      if (Subsumes(s, r) and G(s, r, H) < g)
         remove r from R
Return S
```

R is a set of stochastic rules. D is the set of PDIs observed 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[3]. For an explanation of calculation of the G statistic see [14].

2.2.5 AprioriFilter

The AprioriFilter function is similar to filter, but checks candidate rules at level k against rules at level k-3.

```
AprioriFilter(Ck, L[k-3], significant)
RulesL[k-3] = {l ∈ L[k-3] | HasEffect(l)}
for (c ∈ Ck where HasEffect(c))
   for (lr ∈ RulesL[k-3])
      if (Subsumes(lr,c) and G(c,lr) < significant) {
         remove c from Ck;
         next c;
      }
```

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

The significant parameter defines the g statistic level at which we filter. Rules filtered by this function are removed in the same way as pruned rules, and therefore take no further part in rule generation. For example, if the rule: a ← b is removed by this method no further rules will be generated with head *a* and body *b* (e.g. {a ← b,c}, {a ← b,d}). This 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 significant parameter to 0.1 (3.84 would be 95% significance), by not filtering until rules at level 4 (i.e. the rule {a ← b,c,d} can be filtered by {a ← b,c}, and by filtering against rules with three less conditions (k-3). Further experimentation is required in this area.

The *aprioriFilter* function alters the stopping criteria through removing rules that do not appear significant at each level. Apriori 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.

2.2.6 Generating Rule Probabilities

The rule probability (*prob*), which is the probability of the effect (rule head) being true if the body (conditions) is true is derived empirically as *prob = support/bodySupp*.

2.2.7 Add Rule Complements

The *Filter* function often filters rules and not their complements. For example, the variable, NORTH, can take the values: EMPTY_NORTH, AGENT_NORTH, and WALL_NORTH. The filter process could filter rule 2 below, but leave 1 and 3.

1. 0.6: EMPTY_NORTH ← MOVE_NORTH, EMPTY_WEST
2. 0.1: AGENT_NORTH ← MOVE_NORTH, EMPTY_WEST
3. 0.3: WALL_NORTH ← MOVE_NORTH, EMPTY_WEST

This would cause a problem in the state generation (section 4), because the set of rules will not generate states with AGENT_NORTH present. The algorithm iterates through all rules in the learned dependencies, *R*, checking that all possible values of its effect fluent 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*.

```
AddRuleComplements(R, D)
for (r ∈ R) do
   f = r.head;
   for (fValue ∈ possibleValues(f))
      if (fValue ≠ f.value)
         newRule = copy of f with f.head set to fValue
            if (newRule ∈ R)
               if (newRule matches a PDI in D)
                  add newRule to R
```

3 The Precedence Algorithm

The *Precedence* algorithm provides a method of resolving conflicts when more than one *rule set* is applicable to the same state. A *rule set* is defined as a set of rules that apply to the same output variable and has the same body.

Example: The percept at time t-1 is {EMPTY_NORTH, WALL_EAST, AGENT_SOUTH, EMPTY_UNDER} and the action at time t-1 is MOVE_NORTH. The conflicting rule sets in Table 1 and Table 2 apply to the same output variable, NORTH, which can take values NORTH_WALL, NORTH_EMPTY, NORTH_AGENT. The *Precedence* algorithm defines how conflicts of this nature are resolved.

Table 1. Rule set with body: action = MOVE_NORTH and percept contains EMPTY_NORTH

Effect	*Conditions*
0.6: EMPTY_NORTH	← MOVE_NORTH, EMPTY_NORTH
0.1: AGENT_NORTH	← MOVE_NORTH, EMPTY_NORTH
0.3: WALL_NORTH	← MOVE_NORTH, EMPTY_NORTH

Table 2. Rule set with body: action = MOVE_NORTH and percept contains AGENT_SOUTH

Effect	*Conditions*
0.7: EMPTY_NORTH	← MOVE_NORTH,AGENT_SOUTH
0.3: WALL_NORTH	← MOVE_NORTH,AGENT_SOUTH

The precedence algorithm evaluates the precedence of a generated set of rules *R*, over a set of PDIs, *D*, and proceeds in the following 2 steps:

1. Categorise all rules into *rule sets.* A *rule set* is a group of rules with the same output variable and the same body (section 3.1).
2. For all PDIs in the database, if two rule sets apply to the same PDI and have same output variable, define which one has precedence using the *FirstRuleSetSuperior* function. This function finds the subset of PDIs for which both rule sets apply and uses an error measure to check which set performs best on the subset (section 3.2).

```
Precedence(R, D)
ruleSets = FormRuleSets(R, D)
for (p ∈ D){
   matchedRules = MatchingRules(ruleSets, p);
   for (rSet1 ∈ RuleSetsIn(matchedRules)) {
      for (rSet2 ∈ RuleSetsIn(matchedRules)) {
         if (rSet1 == rSet2) next rSet2;
         if (OutputVar(rSet1) ≠ OutputVar(rSet2)) next rSet2;
         if (PrecedenceSet(rSet1, rSet2) next rSet2;
         if (FirstRuleSetSuperior(set1, set2))
            SetPrecedenceOver(set2, set1);
         else
            SetPrecedenceOver(set1, set2);
}  }  }
```

The *MatchingRules* function returns the subset of rule sets with a body matching the percept and action from the PDI. The *RuleSetsIn* function returns the rule sets contained in the matching rules. The *OutputVar* function returns the variable that forms the head of a rule (rather than it's actual value). If the head of a rule is EMPTY_NORTH then the variable is NORTH.

Note that *D* can be either the same set of data used to learn the rules, or a separate set used purely to test the rules. If the same data set is used, the speed of the algorithm can be increased by the observation that a specific rule set (one with more conditions) will always have precedence over a general one (section 3.2).

3.1 FormRuleSets

Rule sets are sets of rules with the same conditions (rule body) which apply to the same output variable. E.g. {Prob: 0.5: EMPTY_NORTH ← MOVE_NORTH, Prob: 0.5: AGENT_NORTH ← MOVE_NORTH} is a rule set for the variable NORTH.

```
FormRuleSets(R)
for (r ∈ R) {
   if (NotEmpty(r.ruleSet) next r;
   for (c ∈ R) {
      if (Body(r) ≠ Body(c)) next c;
      if (OutputVar(r) ≠ OutputVar(c)) next c;
      if (r ∈ c.ruleSet)) {
         copy(r.ruleSet, c.ruleSet)); next r; }
      r.ruleSet += c;
}}
```

3.2 FirstRuleSetSuperior

The *FirstRuleSetSuperior* function returns true if the first rule set should have precedence in situations where the two rule sets conflict. This is achieved by comparing the probability values for the output variable of the rule sets with a new rule set generated by combining their conditions. The probabilities for the new rule set are generated empirically in the same manner as all other rules (section 2.2.6) *prob = support/bodySupp*.

Example: The rule sets in Table 1 and Table 2 have the conditions {MOVE_NORTH, EMPTY_NORTH} and {EMPTY_NORTH, AGENT_SOUTH} respectively. If the two sets of conditions are combined and the effects (rule heads) added the new rule set shown in Table 3 is generated.

Table 3. Rule set formed from the combination of rule sets in Table 1 and Table 2.

0.75: EMPTY_NORTH	← MOVE_NORTH, MOVE_NORTH, EMPTY_NORTH
0.0: AGENT_NORTH	← MOVE_NORTH, EMPTY_NORTH, AGENT_SOUTH
0.25: WALL_NORTH	← MOVE_NORTH, EMPTY_NORTH, AGENT_SOUTH

The rule set that has the least error when compared to the combined rule set is given precedence over the other. In the implementation used for this paper, an error of +0.5 was given for each non-matching output and the difference otherwise. The rule set in Table 1 would therefore have an error of 1.0 (for AGENT_NORTH) + (0.75 – 0.6 = 0.15) for v_1 + (0.3 – 0.25 = 0.05). The total error is therefore 0.7. This error measure is somewhat arbitrary, but was defined in order to penalise rules which failed to generate all values for a variable, however infrequently that variable occurs.

Note 1: A rule set which is *subsumed* by a more general rule will always have precedence over a specific 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 are {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.

4 Generating States from learned rules

The state generator function 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 ASDD from the history of observations. The generated states can then be used as a model by a reinforcement learning algorithm such as value learning to generate a policy. This method has been applied previously in [4].

Our implementation of the ASDD algorithm generates a set of rules with only one fluent in the effects in order to reduce substantially the number of rules that must be evaluated. States are generated as follows:

1. Find all rules matching the current state and selected action. This is the subset of rules with conditions matching the state and action.
2. Remove rules that defer to other matching rules. For each rule in the rule set from step 1, remove if another rule has precedence over it.
3. Generate possible states and probabilities (section 4.1).
4. Remove impossible states using constraints and normalise the state probabilities (section 4.2).

4.1 Generate Possible States

The possible states are generated as follows:

1. Create a new state from each combination of effect fluent values in the rules remaining after steps 1 and 2 above.
2. Multiply the probability of each effect rule to generate the probability of each state.

In order to demonstrate this process, we refer back to the predator-prey scenario, introduced in section 1.1, which forms the basis of the experiments in section 5 and shows how the predator generates states from the learned rules.

After steps 1 and 2 from section 4, we are left with the rules in **Table 4** for the initial percept {WALL_NORTH, EMPTY_EAST, EMPTY_SOUTH, AGENT_WEST, EMPTY_UNDER} and action MOVE_NORTH.

Table 4. Rules generated by the ASDD algorithm for the predator prey scenario matching the initial percept WALL_NORTH, EMPTY_EAST, EMPTY_SOUTH, AGENT_WEST, EMPTY_UNDER and action MOVE_NORTH, after removal of rules by precedence.

Prob:	*Effect*	*Conditions*
1.0	WALL_NORTH	MOVE_NORTH, WALL_NORTH
1.0	EMPTY_EAST	MOVE_NORTH, EMPTY_EAST, AGENT_WEST
1.0	EMPTY_SOUTH	MOVE_EAST, WALL_EAST
0.59	EMPTY_WEST	MOVE_NORTH, EMPTY_EAST, AGENT_WEST
0.41	AGENT_WEST	MOVE_NORTH, EMPTY_EAST, AGENT_WEST
0.63	EMPTY_UNDER	MOVE_NORTH, WALL_NORTH, AGENT_WEST
0.37	AGENT_UNDER	MOVE_NORTH, WALL_NORTH, AGENT_WEST

The states generated from the rules in Table 4 are shown in Table 5. The probabilities for each state are generated by multiplying the probabilities of each rule that generated the state.

Table 5. Generated states and associated probabilites from the rules in Table 4

WALL_NORTH	EMPT_EAST	EMPT_SOUTH	EMPT_WEST	EMPT_UNDER	Pr: 0.37
WALL_NORTH	EMPT_EAST	EMPT_SOUTH	EMPT_WEST	AGEN_UNDER	Pr: 0.22
WALL_NORTH	EMPT_EAST	EMPT_SOUTH	AGEN_WEST	EMPT_UNDER	Pr: 0.25
WALL_NORTH	*EMPT_EAST*	*EMPT_SOUTH*	*AGEN_WEST*	*AGEN_UNDER*	*Pr: 0.15*

There were two rules for the *west* variable with results EMPTY and AGENT, and two rules for the *under* 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.

4.2 Removing 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 percept with two agents when there is only one agent in the world (e.g. the rule in Italics in Table 5). Ultimately, the agent should generate it's own constraints that define impossible world states. A rule such as IMPOSSIBLE (AGENT_NORTH, AGENT_SOUTH) allows elimination of the impossible world states generated. 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, as it is too far detached from the real world states. Currently our system removes impossible states by checking that each generated state contains only one agent, and does not have walls opposite each other, but the *impossible* function should be simple to create by observing rule element sets eliminated in the *prune* step of the ASDD algorithm.

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.

Despite the addition of the two constraints mentioned, the state generator is still able to generate erroneous states as is demonstrated below. Removing states of this type is a complex problem as the states themselves are not impossible.

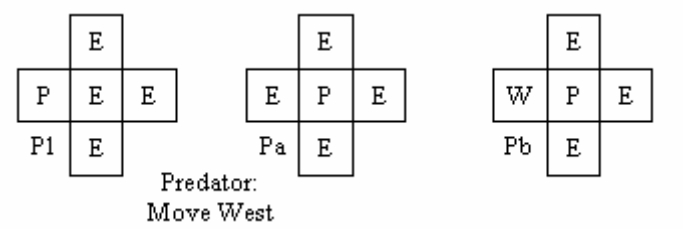

Fig. 2. : Generation of erroneous states. From the initial state P1 in which the prey is immediately to the west, the state generator generates the states Pa and Pb after a move west action. Situation Pa is in fact not possible, because the predator and prey take simultaneous moves. For the predator to be on top of the prey after a move west, the prey would have to have stayed still. This is only possible if it moved into a wall, which it cannot have done as all the square around it are empty.

5 Results

Table 6 compares the speed of ASDD against the MSDD algorithm. Timings were taken on learning rules from data sets of 100 to 20000 observations of random moves. Performance was measured on a 350MHz Pentium with 256MB RAM. Although these are only preliminary tests, we found that ASDD displayed roughly equal performance to MSDD initially, and that the time taken to learn rules increased roughly in proportion to the size of the data set for MSDD. On larger data sets time taken by ASDD starts to level and thus shows a dramatic performance increase against MSDD for 20000 observations. ASDD minimum support was set to 1 (any occurrence means a rule set is not discarded), and significant in AprioriFilter to 0.1. For both ASDD and MSDD g in *Filter* was set to 1.17.

Table 6. Time taken (in seconds) to learn rules with data collected from 100, 1000, 2000, 5000, 10000 and 20000 random moves.

	100	*1000*	*2000*	*5000*	*10000*	*20000*
ASDD	36	227	303	471	641	828
MSDD	12	213	442	1151	2363	4930

Table 7 gives an 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 200,000 trials (a “correct” state map). The state map records, for each percept and action, the relative frequencies of each next percept. The error measure is defined as follows: For each state generated which is not present in the “correct” state map add 0.5 to the error. For each in the “correct” state map which is not in the generated set, add 0.5 to the error. If both state sets contain the same state add the difference in probability for the two states. The total number of state-action pairs in the “correct” map was 168 and total state-action following states was 852.

Table 7. Error measure of generated states generated from rules learned from data collected over 100, 1000, 2000, 5000, 10000 and 20000 random moves.

	100	*1000*	*2000*	*5000*	*10000*	*20000*
State Map	415.9	355.2	261.8	135.1	40.5	15.3
ASDD	480.0	335.1	274.6	220.4	197.9	108.9
MSDD	482.7	333.5	280.5	198.7	137.6	92.18

The performance of both rule-learning methods is poor against a state map generated from the same number of trials except in 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 generating states. This indicates that the error lies in our state generation process, rather than the rules themselves. Further investigation is required to discern why the error rate is high for both rule sets. A possible reason for the error is the removal of impossible states problem outlined in section 4.2.

6 Conclusions

This paper presents the ASDD algorithm, which is the first step in the development of an efficient algorithm for learning stochastic logic rules from data. Results in our preliminary tests are extremely encouraging in that the algorithm is able to learn rules accurately and at over twice the speed of MSDD. Future extensions to the method are expected to greatly increase the performance and application of the algorithm. Some initial areas to examine are:

- Increasing the performance of the algorithm by use of efficient subset function and transaction ID approaches from Apriori.
- Testing the algorithm on a wide variety of data sets to give a better performance measure.
- Implementing incremental updating of rules using methods from association rule mining.
- Generating first-order rules from data through the inclusion of background knowledge.

Acknowledgements:

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

References

1. Agrawal, R. and Srikant, R.: Fast Algorithms for Mining Association Rules. In: *Proc. 20th Int. Conf. Very Large Data Bases {VLDB}*, 12-15, Bocca, J.B., Jarke, M., Zaniolo, C. (eds.), Morgan Kaufmann, (1994).
2. Boutilier, C. and Dean, T. and Hanks, S. Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research* 11: 1-94. (1999).

3. Cheung, D.W. and Han, J., Ng, V., and Wong, C.Y., Maintenance of discovered association rules in large databases: An incremental updating technique. In Proc. 1996 Int. Conf. Data Engineering, pages 106--114, New Orleans, Louisiana, Feb. (1996)
4. Child, C. and Stathis, K. 2003. SMART (Stochastic Model Acquisition with ReinforcemenT) Learning Agents: A Preliminary Report. Adaptive Agents and Multi-Agent Systems AAMAS-3. AISB 2003 Convention., Dimitar Kazakov. Aberystwyth, University of Wales. ISBN 1 902956 31 5. (2003).
5. Drescher, G.L. *Made-Up Minds, A Constructivist Approach to Artificial Intelligence*. The MIT Press. (1991)
6. Fikes, R.E. and Nilsson, N.J. STRIPS: a new approach to the application of theorem proving to problem-solving. *Artificial Intelligence* 2(3-4): 189-208 (1971).
7. Hidber C., Online Association Rule Mining. SIGMOD Conf., 1999. http://citeseer.nj.nec.com/hidber98online.html (1999).
8. Hipp, J. and Gunter, U. and Nakhaeizadeh, G., Algorithms for Association Rule Mining - A General Survey and Comparison, *SIGKDD Explorations*, 2000, vol. 2, no. 1, 58-64, July. (2000).
9. Kaelbling, L.P. and Littman, H.L. and Moore, A.P. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research* 4: 237-285. (1996)
10. Kaelbling, L. P., and Oates, T. and Hernandez, N. and Finney, S. Learning in Worlds with Objects, citeseer.nj.nec.com/kaelbling01learning.html.
11. McCarthy, J. and Hayes, P.J. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4: 463-502. (1969).
12. Muggleton, S.H. Learning Stochastic Logic Programs. *Proceedings of the AAAI2000 Workshop on Learning Statistical Models from Relational Data*, L. Getoor and D. Jensen, AAAI. (2000).
13. Murphy, K.P. Dynamic Bayesian Networks: Representation, Inference and Learning. Ph.D. Thesis, University of California, Berkeley. (2002).
14. Oates T., Schmill, M.D., Gregory, D.E. and Cohen P.R. Detecting complex dependencies in categorical data. Chap. in *Finding Structure in Data: Artificial Intelligence and Statistics V*. Springer Verlag. (1995).
15. Oates, T. and Cohen, P. R.Learning Planning Operators with Conditional and Probabilistic Effects. *AAAI-96 Spring Symposium on Planning with Incomplete Information for Robot Problems*, AAAI. (1996).
16. Sutton, R.S., and A.G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, MIT Press. (1998).