ARMS: Action-Relation Modelling System for Learning Action Models

Kangheng Wu [1,2], **Qiang Yang**[1] **and Yunfei Jiang**[2]
[1]Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong
[2]Software Institute, Zhongshan University, Guangzhou, China
khwu@cs.ust.hk, qyang@cs.ust.hk and lncsri05@zsu.edu.cn

Abstract

We present a system for automatically discovering action models from a set of successful observed plans. AI planning requires the definition of an action model using a language such as PDDL as input. However, building an action model from scratch is a difficult and time-consuming task even for experts. Unlike the previous work in action-model learning, ARMS does not assume complete knowledge of states in the middle of the observed plans; in fact, our approach would work when no or partial intermediate states are given. These example plans are obtained by an observation agent who does not know the logical encoding of actions and the full state information between actions. In a real world application, the cost is prohibitively high in labelling the training examples by manually annotating every state in a plan example from snapshots of an environment. To learn action models, ARMS gathers knowledge on the statistical distribution of frequent sets of actions in the example plans. It then builds a weighted propositional satisfiability (Weighted SAT) problem and solves it using a weighted MAX-SAT solver. We lay the theoretical foundations of the learning problem and evaluate the effectiveness of ARMS empirically.

Introduction

AI planning systems require the definition of an action model, an initial state and a goal to be provided as input. In the past, various action modelling languages have been developed. Some examples are the STRIPS language (Fikes & Nilsson 1971), ADL language (Pednault 1986) and PDDL language (Ghallab *et al.* ; Fox & Long 2003). With these languages, a domain expert sits down and analyzes a collection of the observed plans, then writes a complete set of domain action representations. These representations are then used by planning systems as input to generate plans.

However, building an action model from scratch is a task that is exceedingly difficult and time-consuming even for domain experts. Because of the difficulty, various approaches (Wang 1995) have been explored to learn action models from plans. A common feature of these works is requiring states just before or after each action to be known. Statistical and logical inferences can then be made to learn actions' preconditions and effects.

In this competition, we take one step toward in the direction of learning action models from observed plans with partial information. The resultant algorithm is called ARMS , which stands for *Action-Relation Modelling System* (Yang, Wu, & Jiang 2005). We assume that each observed plan consists of a sequence of action names together with their parameters and the initial states and goal conditions. We assume that the intermediate states between actions are only partially known; that is, between every adjacent pair of actions, the truth of a literal can be unknown. Thus, it is possible that any intermediate state is completely unknown to us. Suppose that we have several observed plans as input. From this incomplete knowledge, our system automatically guesses a minimal logical action model that can explain most of the observed plans. This action model is not guaranteed to be completely correct, but we hope that it serves as an add-on component for the knowledge editors which can provide advice for human users, such as GIPO (McCluskey, Liu, & Simpson 2003)

Consider a simple example in the `depots` problem, as three observed plans in Table 1. Any literal such as $(on\ c0\ p0)$ in the goal conditions might be established by the first action, the second, or any of the rest. We wish to learn the preconditions, add and delete lists of all actions to explain these observed plans. It is this uncertainty that gives difficulty to previous approaches. In contrast to the previous approaches, from these example plans, our algorithm generates a good possible model in which the actions' preconditions, add and delete lists are filled. An example output for the $(load\ x\ y\ z\ p)$ operator is:

action	load(x - hoist y - crate z - truck p - place)
pre:	(at x p), (at z p), (lifting x y)
add:	(in y z), (available x)
del:	(lifting x y)

The design of ARMS is motivated by the fact that in many cases, it is possible for an onlooker to observe and record the sequences of agents' action names in a task domain, but to encode these actions logically is much harder. An agent's plans can be automatically recorded by various sensors attached to the agent and by a sensor network distributed in the environment. Plans can also be made avail-

Table 1: Three plan examples

	Plan1	Plan2	Plan3
Initial	I_1	I_2	I_3
1	(lift h1 c0 p1 ds0), (drive t0 dp0 ds0)	(lift h1 c1 c0 ds0)	(lift h2 c1 c0 ds0)
State		(lifting h1 c1)	
2	(load h1 c0 t0 ds0)	(load h1 c1 t0 ds0)	(load h2 c1 t1 ds0)
3	(drive t0 ds0 dp0)	(lift h1 c0 p1 ds0)	(lift h2 c0 p2 ds0), (drive t1 ds0 dp1)
State	(available h1)		
4	(unload h0 c0 t0 dp0)	(load h1 c0 t0 ds0)	(unload h1 c1 t1 dp1), (load h2 c0 t0 ds0)
State	(lifting h0 c0)		
5	(drop h0 c0 p0 dp0)	(drive t0 ds0 dp0)	(drop h1 c1 p1 dp1), (drive t0 ds0 dp0)
6		(unload h0 c1 t0 dp0)	(unload h0 c0 t0 dp0)
7		(drop h0 c1 p0 dp0)	(drop h0 c0 p0 dp0)
8		(unload h0 c0 t0 dp0)	
9		(drop h0 c0 c1 dp0)	
Goal State	(on c0 p0)	(on c1 p0) (on c0 c1)	(on c0 p0) (on c1 p1)

I_1 : (at p0 dp0), (clear p0), (available h0), (at h0 dp0), (at t0 dp0), (at p1 ds0), (clear c0), (on c0 p1), (available h1), (at h1 ds0)
I_2 : (at p0 dp0), (clear p0), (available h0), (at h0 dp0), (at t0 ds0), (at p1 ds0), (clear c1), (on c1 c0), (on c0 p1), (available h1), (at h1 ds0)
I_3 : (at p0 dp0), (clear p0), (available h0), (at h0 dp0), (at p1 dp1), (clear p1), (available h1), (at h1 dp1), (at p2 ds0), (clear c1), (on c1 c0), (on c0 p2), (available h2), (at h2 ds0), (at t0 ds0), (at t1 ds0)

able through a textual descriptions of the steps in a task domain; for example the Web site *www.ehow.com* provides step-by-step instructions on how to fix an automobile, and *www.epicurious.com* provides step-by-step recipes. Finally, the plan examples can be provided by a domain expert, who may find it much easier to simply inform us what to do, without telling us the precise reasons for why each step is taken. It is thus intriguing to ask whether we could intelligently guess, or approximate, an action model in an application domain if we are only given a set of recorded action occurrences but partial or even no intermediate state information. In this competition, we take a first step to answer this question by presenting a learning system.

ARMS proceeds in two phases. In phase one of the algorithm, ARMS finds the frequent action sets from plans that share a common set of parameters. In addition, ARMS finds some frequent predicate-action relations with the help of the initial state, the goal state and the partial intermediate states. These predicate-action relations give us an initial guess at the preconditions, add lists and delete lists of actions in this subset. These action subsets and relations are used to obtain a set of constraints that must hold in order to make the plans correct. In phase two, we transform the constraints extracted from the plans into a weighted SAT representation (Borchers & Furman 1999), solve it, and produce an action model from the solution of the SAT problem. The process iterates until all actions are modelled.

For a reasonably large set of training observed plans, the corresponding SAT representation is likely to be too complex to be solved efficiently. In response, we provide a heuristic method for modelling the actions approximately, and measure the correctness of the model using the error and redundancy rates. We present a cross-validation method for evaluating the learned action model against different experimental parameters such as the size and the number of plans.

The rest of the paper is organized as follows. The next section discusses related work in more detail. This is followed by the algorithm description. In the experiments section, we develop a cross-validation evaluation strategy for our learned action models and test our algorithm in several different domains. We conclude in the last section with a discussion of future work.

Related Work

The problem of learning action descriptions is important in AI Planning. As a result, several researchers have studied this problem in the past. In (Wang 1995), (Oates & Cohen 1996) and (Gil 1994), a partially known action model is improved using knowledge of intermediate states between pairs of actions in a plan. These intermediate states provide knowledge for which preconditions or post-conditions may be missing from an action definition. In response, revision of the action model is conducted to *complete* the action models. In (Wang 1995) an STRIPS model is constructed by computing the most specific condition consistent with the observed examples. In (Shen 1994) a decision tree is constructed from examples in order to learn preconditions. However, these works require there to be an incomplete action model as input, and learning can only be done when the intermediate states can be observed.

In (Sablon & Bruynooghe 1994) an inductive logic programming (ILP) approach is adopted to learn action models. Similarly, in (Benson 1995), a system is presented to learn the *preimage* or precondition of an action for a *TOP* operator using ILP. The examples used require the positive or negative examples of propositions held in states just before each action's application. This enables a concept for the preimage of an action to be learned for each state just before that action. ILP can learn well when the positive and negative examples of states before all target actions are given. Even though one can still use logical clauses to enumerate the different possibilities for the precondition, the number of such possibilities is going to be huge. Our SAT-based solution provides an elegant control strategy for resolving ambiguities within such clauses.

Another related thrust adopts an approach of knowledge acquisition, where the action model is acquired by interacting with a human expert (Blythe *et al.* 2001). Our work can be seen as an add-on component for such knowledge editors which can provide advice for human users, such as GIPO (McCluskey, Liu, & Simpson 2003). The DISTILL system (Winner & Veloso 2002) which is a method to learn program-like plan templates from example plans and shares similar motivation with our work.

In propositional logic, the satisfiability (SAT) problem is aims to find an assignment of values to variables that can satisfy a collection of clauses. If it fails to do so, the SAT solvers will give an indication that no such assignment exits (Moskewicz *et al.* 2001; Kautz & Selman 1996; Zhang 1997). Each clause is a disjunction of literals, where each of which is a variable or its negation. There are many local search algorithms for the satisfiability problem. The weighted MAX-SAT solvers assign a weight to each clause and seeks an assignment that maximizes the sum of the weights of the satisfied clauses (Borchers & Furman 1999).

The ARMS Algorithm

In this competition, we will learn a PDDL (level 1, STRIPS) representation of actions from the observed plans. The input to the algorithm is a set of the observed plan examples, where each plan consists of (1) a list of propositions in the initial state, (2) a list of propositions that hold in the goal state, (3) a list of partial intermediate observed states, and (4) a sequence of observed action names and instantiated parameters. For simplicity, we use α_{init} to represent the initial state and α_{goal} to represent the goal state in a plan.

Our ARMS algorithm is iterative, where in each iteration, a selected subset of actions in the training plan examples are learned. An overview of our algorithm is as follows:

Step 1 We first convert all action instances to their schema forms by replacing all constants by their corresponding variable types. Let Λ be the set of all incomplete action schemata. Initialize the action model Θ by the set of complete action schemata; this set is empty initially if all actions need to be learned.

Step 2 Build a set of predicate and action constraints based on individual actions and call it Ω. Apply a frequent-set mining algorithm to find the frequent sets of *related* actions and predicates (see the next subsection). Here *related* means the actions and predicates must share some common parameters. Let the frequent set of action-predicate relations be Σ.

Step 3 Build a weighted maximum satisfiability representation Γ based on Ω and Σ.

Step 4 Solve Γ using a weighted MAXSAT solver [1].

Step 5 Select a set A of solved actions in Λ with the highest weights. Update Λ by $\Lambda - A$. Update Θ by adding A. Update the frequent sets Σ by removing all relations involving only complete action models. Update the initial states of all plan examples by executing the action set A. If Λ is not empty, go to Step 2.

[1]http://www.nmt.edu/ borchers/maxsat.html

Step 6 Output the action model Θ.

The algorithm starts by initializing a set of action schemata yet to be explained. Subsequently, it iteratively builds a weighted MAXSAT representation and solves it. Each time a few more actions are explained, which are used to build new initial states. ARMS terminates when all actions in the example plans are learned. As a result, the actions of all plans examples are learned in a left-to-right sweep if we assume the plans begin on the left and end on the right, without skipping any incomplete actions in between. In this way, every time Step 2 is re-encountered, a new set of initial states is constructed by executing the newly learned actions in their corresponding initial states.

Step 1: Initialize Plans and Variables

An observed plan example consists of a set of proposition and action instances. Each action instance represents an action with some instantiated objects. We *convert* all plans by substituting all occurrences of an instantiated object in every action instance with the same variable parameter. If the object has multiple types, we generate a clause to represent each possible type for the object. For example, if an object o has two types $Block$ and $Table$, the clause becomes: $\{(o = Block)\ or\ (o = Table)\}$. The two nearby actions are connected with the parameter-connector set, which is a list of pairs linking a parameter in one action to a parameter in another action.

Step 2: Build Action, Information and Plan Constraints

A weighted MAXSAT consists of a set of clauses representing their conjunction(Borchers & Furman 1999). Each clause represents a constraint that should be satisfied. The weights associated with a clause represents the degree in which we wish the clause to be satisfied. Given a weighted MAXSAT problem, a SAT solver finds a solution by maximally satisfying all the clauses with high weight values. In the ARMS system, we have four kinds of constraints to satisfy, representing four types of clauses. They are predicate, action, information and plan constraints. The predicate constraints are derived from statistics directly (see Step 3).

Action Constraints The *second* type of clauses in a SAT represents the requirement of individual actions. These constraints make the learning problem reasonable for the general action definitions, but not for the special cases. Here we define a predicate to be *relevant* to an action when they share a parameter. Let pre_i, add_i and del_i represent a_i's precondition list, add-list and del-list. These *general action constraints* are translated into the following clauses:

1. (Constraint A.1) Every action a_i in a plan must add a relevant predicate for the precondition of a later action a_j in the plan at least; this predicate is known as a primary effect of an action (Fink & Yang 1997). If a predicate does not share any parameter with any primary effect and any precondition list of action goal, then the predicate is not in its add and delete lists. Let $par(p_k)$ be the predicate p_k's parameters and pri_i be the a_i's primary effects.

$(par(p_k) \cap par(pri_i) = \phi) \wedge (par(p_k) \cap par(pre_{goal}) = \phi) \Rightarrow p_k \notin add_i \wedge p_k \notin del_i$

2. (Constraint A.2) Every action's add list cannot negate predicates that appear in the precondition list. The intersection of the precondition and add lists of all actions must be empty.
   $pre_i \cap add_i = \phi$.
3. (Constraint A.3) If an action's del list includes a predicate, this predicate is assumed to be needed by an instance of the action, that is, an action's precondition includes the predicate.
   $del_i \subseteq pre_i$.

Example 1 *Consider the action (load x - hoist y - crate z - truck p - place) in Table 1, the possible predicates of the action (load x y z p) are (at x p), (available x), (lifting x y), (at y p), (in y z), (clear y), and (at z p). The precondition list pre_{goal} of action goal consists of (on y - crate s - surface). Suppose that its primary effect is (in y z). From this action (load x y z p), the SAT clauses are as follows:*

- *A possible precondition list includes all of the possible predicates that are joined by a disjunction. From $A.1$ above, the possible predicates of the add and delete list are (lifting x y), (at y p), (in y z), (clear y), or (at z p).*
- *$A.2$ can be encoded as the conjunction of the following clauses:(1) (lifting x y)$\in add_i \Rightarrow$ (lifting x y)$\notin pre_i$, (2) (at y p)$\in add_i \Rightarrow$ (at y p)$\notin pre_i$, (3) (in y z)$\in add_i \Rightarrow$ (in y z)$\notin pre_i$, (4) (clear y)$\in add_i \Rightarrow$ (clear y)$\notin pre_i$, (5) (at z p)$\in add_i \Rightarrow$ (at z p)$\notin pre_i$, (6) (lifting x y)$\in pre_i \Rightarrow$ (lifting x y)$\notin add_i$, (7) (at y p)$\in pre_i \Rightarrow$ (at y p)$\notin add_i$, (8) (in y z)$\in pre_i \Rightarrow$ (in y z)$\notin add_i$, (9) (clear y)$\in pre_i \Rightarrow$ (clear y)$\notin add_i$, (10) (at z p)$\in pre_i \Rightarrow$ (at z p)$\notin add_i$.*
- *$A.3$ can be encoded as the conjunction of the following clauses: (1) (lifting x y)$\in del_i \Rightarrow$ (lifting x y)$\in pre_i$, (2) (at y p)$\in del_i \Rightarrow$ (at y p)$\in pre_i$, (3) (in y z)$\in del_i \Rightarrow$ (in y z)$\in pre_i$, (4) (clear y)$\in del_i \Rightarrow$ (clear y)$\in pre_i$, (5) (at z p)$\in del_i \Rightarrow$ (at z p)$\in pre_i$*

Information Constraints The *third* type of constraint is used to explain why the observed intermediate states exist in a plan. Suppose we observe a proposition p between two actions a_n and a_{n+1}, and p, a_{i1},..., and a_{ik} share the same constant parameters, we can present the facts by the following clauses. a_{i1},...,and a_{jk} appear in the plan's partial order.

- (Constraint I.1) The predicates p must be generated by one action $a_{ik}(0 \leq ik \leq n)$ at least, that is, p is selected to be in the add-list of a_{ik}. $p \in (add_{i1} \cup add_{i2} \cup ... \cup add_{ik})$
- (Constraint I.2) The last action $a_i k$ must not delete the predicate p, that is, p must not be selected to be in the del list of $a_i k$. $p \notin del_{ik}$

Example 2 *Consider the intermediate state $(lifting\ h0\ c0)$ in Plan 1 in Table 1, it can be generated by the actions $(lift\ h1\ c0\ p1\ ds0)$, $(load\ h1\ c0\ t0\ ds0)$ or $(unload\ h0\ c0\ t0\ dp0)$. But it cannot be deleted by (unload h0 c0 t0 dp0). The SAT clauses are as follows:*

- (I.1) $(lifting\ x\ y) \in (add_{lift} \cup add_{load} \cup add_{unload})$
- (I.2) $(lifting\ x\ y) \notin del_{unload}$

Plan Constraints The *fourth* type of constraint represents the relationship between actions in a plan to ensure that a plan is correct. Since the relations explain why two actions co-exist, and such pairs are generally overwhelming in a set of plans, as a heuristic we wish to restrict ourselves to only a small subset of frequent action pairs. Therefore, we apply a frequent-set mining algorithm from data mining in order to obtain a set of frequent action pairs and predicate-action triples from the plan examples. In particular, we apply the Apriori algorithm (Agrawal & Srikant 1994) to find the ordered action sequences $< a_i, a_{i+1}, ..., a_{i+n} >$ where $a_{i+j}(0 \leq j \leq n)$ appears in the plan's partial order. The prior probability (also known as *support* in data mining literature) of action sequences $< a_i, a_{i+1}, ..., a_{i+n} >$ in all plan examples is no less than a certain probability threshold θ (known as the minimum support in data mining). We do not suffer the problem of the generating too many redundant association rules as in data mining research, since we only apply the Apriori algorithm to find the frequent pairs; these pairs are to be explained by the learning system later. In the experiments, we will vary the value of θ to verify the effectiveness of algorithm. When consider subsequences of actions from example plans, we only consider those sequences whose supports are over θ.

For each pair of actions in a frequent sequence of actions, we generate a constraint to encode *one* of the following situations. These are called plan constraints:

- (Constraint P.1) One of the relevant predicates p must be selected to be in the preconditions of both a_i and a_j, but not in the delete list of a_i,
  $\exists p\ (p \in (pre_i \cap pre_j) \wedge p \notin (del_i))$
- (Constraint P.2) The first action a_i adds a relevant predicate that is in the precondition list of the second action a_j in the pair,
  $\exists p\ (p \in (add_i \cap pre_j))$
- (Constraint P.3) A relevant predicate p that is deleted by the first action a_i is added by a_j. The second clause is designed for the event when an action re-establishes a fact that is deleted by a previous action.
  $\exists p\ (p \in (del_i \cap add_j))$
  The above three plan constraints can be combined into one constraint $\Phi(a_i, a_j)$ in ARMS . $\Phi(a_i, a_j)$ is restated as:
  $\exists p\ ((p \in (pre_i \cap pre_j) \wedge p \notin (del_i)) \vee (p \in (add_i \cap pre_j)) \vee (p \in (del_i \cap add_j)))$
- (Constraint P.4) In the general case when n is greater than one, for each sequence of actions in the set of frequent action sequences, we generate a constraint to encode the following situation. The conjunction of every constraint from every pair of actions $< a_i, a_{i+1} >$ in the sequence $< a_i, a_{i+1}, ..., a_{i+n} >$ must be satisfied. $\Phi(a_i, a_{i+1}, ..., a_{i+n})$ can be restated as:
  $\Phi(a_i, a_{i+1}) \wedge \Phi(a_{i+1}, a_{i+2}) \wedge ... \wedge \Phi(a_{i+n-1}, a_{i+n})$

Example 3 *Suppose that ((lift x - hoist y - crate z - surface p - place), (load x - hoist y - crate z - truck p - place), 0) is a* *frequent pair. The relevant parameter is x-x, y-y, p-p. Thus, these two actions are possibly connected by predicates (at x p), (available x), (lifting x y), (at y p), and (clear y). From this pair, the SAT clauses are as follows:*

- *At least one predicate among (at x p), (available x), (lifting x y), (at y p), and (clear y) are selected to explain the connection between (lift x y z p) and (load x y z p).*
- *If $f(x)$ ($f(x)$ can be (at x p), (available x), (lifting x y), (at y p), and (clear y)) connects (lift x y z p) to (load x y z p), then either (a) $f(x)$ is in the precondition list of action (load x y z p) and the add list of (lift x y z p), (b) $f(x)$ is in the delete list of action (lift x y z p) and add list of (load x y z p), or (c) $f(x)$ is in the precondition list of action (lift x y z p), but not in the del list of action (lift x y z p), and in the precondition list of (load x y z p).*

Step 3: Build a Weighted MAXSAT

In solving a weighted MAXSAT problem in Step 3, each clause is associated with a weight value between zero and one. The higher the weight, the higher the priority in satisfying the clause. In assigning the weights to the four types of constraints in the weighted MAXSAT problem, we apply the following heuristics:

1. **Predicate Constraints**. We define the probability of a predicate-action-schema pair as the occurrence probability of this pair in all plan examples. If the probability of a predicate-actin pair is higher than the probability threshold θ, the corresponding constraint receives a weight value equal to its prior probability. If not, the corresponding predicate constraint receives a constant weight of a lower value which is determined empirically (see the experimental section).

   Example 4 *As an example, consider the example in Table 1. A predicate-action pair (clear c0),(lift h1 c0 p1 ds0) (sharing a parameter $\{c0\}$) from Plan1 and predicate-action pair (clear c1),(lift h1 c1 c0 ds0) (sharing a parameter $\{c1\}$) from Plan2 can be generalized to (clear y)$\in pre_{lift}$ (labelled with $\{y\}$). Thus, the support for (clear y), (lift x y z p) with the parameter label y is at least two. Table 2 shows all predicate constraints along with their support values in the previous example.*

Table 2: Examples of All Supported Predicate Constraints

Label	Predicate Constraints	Support
{y}	(clear y)$\in pre_{lift}$	3
{x, p}	(at x p) $\in pre_{lift}$	3
{x }	(available x) $\in pre_{lift}$	3
{z, p}	(at z p) $\in pre_{lift}$	3
{y, p}	(at y p) $\in pre_{lift}$	3
{y, z}	(on y z) $\in pre_{lift}$	3
{x, y}	(at x y) $\in pre_{drive}$	1
{y, z}	(on y z)$\in add_{drop}$	3

2. **Action Constraints**. Every action constraint receives a constant weight. The constant assignment is empirically determined too, but they are generally higher than the predicate constraints.
3. **Information Constraints**. Every partial information constraint receives a constant weight. The constant assignment is empirically determined too, but they are generally highest in the constraints' weights.
4. **Plan Constraints**. The probability of a plan constraint, which is higher than the probability threshold (or a minimal support value) θ, receives a weight equal to its prior probability. We do not suffer the problem of the generation too many redundant association rules as in data mining research, since we only apply the Apriori algorithm to find the frequent pairs; these pairs are to be explained by the learning system later.

Example 5 *Consider the example in Table 1. An action sequence (lift h1 c0 p1 ds0), (load h1 c0 t0 ds0) (sharing parameters $\{$h1, c0, ds0 $\}$) from Plan1 and action sequence (lift h1 c1 c0 ds0), (load h1 c1 t0 ds0) (sharing parameters $\{$h1, c1, ds0 $\}$) from Plan2 can be generalized to (lift x y z p), (load x y z p) (labelled with $\{$y-y, p-p$\}$). The connecter $\{$y-y, p-p$\}$ represents the two parameters y and p in action (lift x y z p) are the same as the two parameters y and p in action (load x y z p), respectively. Thus, the support for (lift x y z p), (load x y z p) with the parameter label $\{$y-y, z-z, p-p$\}$ is at least two. Table 3 shows all plan constraints along with their support values.*

Table 3: Examples of All Action Constraints

Label	Plan Constraints	Support
{y-y, p-p}	Φ(lift, load)	5
{z-x, p-y}	Φ(load, drive)	4
{x-z, z-p}	Φ(drive, unload)	4
{y-y, p-p}	Φ(unload, drop)	5
{x-x, p-p}	Φ(load, lift)	2
{x-x, p-p}	Φ(drop, unload)	1
{x-z, z-p}	Φ(drive, load)	1
{y-p}	Φ(drive, load)	1
{p-p-y }	Φ(lift, load, drive)	4
{z-x-z }	Φ(load, drive, unload)	4
{z-p-p }	Φ(drive, unload, drop),	4
{p-p-p-y}	Φ(load, lift, load, drive)	2
{z-p-p-p }	Φ(drive, unload, drop, unload)	1

Step 5: Update Initial States and Plans

To learn more about the heuristic weights for the predicate constraints in the weighted SAT problem, ARMS obtains frequent sets of related action-predicate pairs. So ARMS select a set A of solved actions in the set of all incomplete actions Λ with the highest weights. It then updates Λ by $\Lambda - A$, and updates Θ by adding A. It then updates the initial states of all plan examples by executing the learned actions in set Θ.

Properties of The ARMS Algorithm

Given a set of observed plans, ARMS can find a large number of models, each making all the example plans in the training set correct. However, we would prefer ARMS to favor a model that is *correct* and *simple* among all models. Since ARMS gets the solution with the heuristic weight, the model is simple.

Definition 1 *A plan is said to be* correct *if (1) all actions' preconditions hold in the state just before that action and (2) all goal propositions hold after the last action.*

Theorem 1 *The solution to a SAT problem formulated using the information, predicate, action and plan constraints, when the probability threshold is set to zero, corresponds to a correct action model.*

Proof Sketch: Suppose a solution is generated using the information, action and plan constraints. Then for each precondition (including goals) of each action in the example plans, there must exist an action before it that achieves this precondition with no actions in between that delete the precondition (plan constraint). In addition, every observed partial state literal is explained through the information constraints. Thus, the plan must be correct.

Error Rate and Redundancy Rate While it is important to guarantee the correctness of a learned action model on the training plans, ARMS uses a greedy algorithm by explaining only the sequences that are sufficiently frequent (with probability greater than θ). Thus, it is still possible that some preconditions of actions in the learned model are not explained, if the preceding actions do not appear frequently enough. Thus, there is a chance that some preconditions remain unexplained.

We define *errors* of an action model M in an observed plan P. The error rate E_c is obtained by computing the proportion of preconditions that cannot be established by any action in the previous part of the plan. This error is used to measure the correctness of the model on the test plans. Similarly, the redundancy rate E_r is calculated as the proportion of predicates in the actions's add list that does not establish any preconditions of any actions in later part of the plan. This error is used to measure the degree of redundancy created by the action model in the test data set. These two rates can be extended to a set of plans by taking P to be the set of plan examples. These two errors together give a picture of how well our learned model explains each test or training plans. Based on this, we can judge the quality of the models.

Complexity Because the process of generating the constraints is linear, The complexity of ARMS mainly depends on the number of clauses and variables in the SAT problem. Consider a domain, there are a actions and n plans. Let P (E) be the bound on the number of the relations in the preconditions (effects) in one action, respectively. If each relation is encoded as a variable in the SAT problem, the bound of the number of the total variables is $O(a * (P + E))$. Let L (C) be the bound on the number of actions (constraints) in each plan. If each constraint is encoded as a clause in the SAT problem, the bound on the number of the total clauses is $O(n * L * C)$.

Experimental Results

It is necessary to assess the effectiveness of the learned action model empirically. Any learned model might be incomplete and error-prone. A first evaluation method (Garland & Lesh 2002) amounts to applying a planning system to the incomplete action model, and then assessing the model's degrees of correctness. In this work, we have an available set of example plans that can serve the dual purpose of training and testing. Thus, we adopt an alternative evaluation method by making full use of all available example plans. We borrow from the idea of cross validation in machine learning. We split the observed plan examples into two separate sets: a training set and a testing set. The training set is used to build the model, and the testing set for assessing the model. We take each test plan in the test data set in turn, and evaluate whether the test plan can be fully *explained* by the learned action model.

In order to evaluate ARMS , we get the observed plans from the planning domains in International Planning Competition 2002 [2], and the plans are generated using the *MIPS* planner [3]. In each domain, we first generated 200 plan examples. We then applied a five-fold cross-validation, where we select *160* plan examples as the training set from four folds, and use the remaining fold with *40* separate plan examples as the test set. The average length of these plans is *50*. We ran all of our experiments on a personal computer with a 768M memory and a Pentium Mobile Processor 1.7GHz CPU using the Linux operating system. We have done tests on five domains, which are described in Table 4. The learned action model is described in Table 5 in the Driverlog domain.

Number of Plans In Figure 1, they are trained on the probability threshold $\theta = 80\%$ and the threshold of partial infor-

Table 4: Five Domains Results(Probability Threshold$\theta = 80\%, PartialInformation = 5\%$)

Domain Name	E_c	E_r	CPU Time (Sec.)
Depots	1%	23%	6
Driverlog	0%	6%	43
Zenotravel	1%	6%	6
Rover	11%	41%	386
Satellite	13%	10%	6

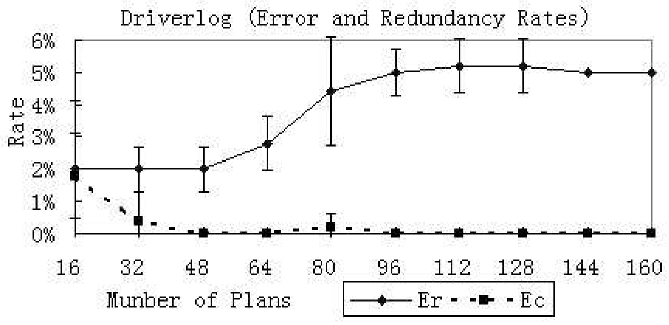

Figure 1: Number of Plans

[2]http://planning.cis.strath.ac.uk/competition/

[3]http://www.informatik.uni-freiburg.de/~mmips/

Table 5: The Learned Action Model(Driverlog Domain, Probability Threshold$\theta = 80\%, PartialInformation = 5\%$)

ACTION	load-truck (obj - obj truck - truck loc - location)
PRE:	(at truck loc), (at obj loc)
ADD:	(in obj truck)
DEL:	(at obj loc)
ACTION	unload-truck(obj - obj truck - truck loc - location)
PRE:	(at truck loc), (in obj truck),
ADD:	(at obj loc)
DEL:	(in obj truck)
ACTION	broad-truck(driver - driver truck - truck loc - location)
PRE:	(at truck loc),(at driver loc),(empty truck),
ADD:	(driving driver truck)
DEL:	(empty truck),(at driver loc)
ACTION	disembark-truck(driver - driver truck - truck loc - location)
PRE:	(at truck loc),(driving driver truck)
ADD:	(at driver loc),(empty truck)
DEL:	(driving driver truck)
ACTION	drive-truck(truck - truck loc-from - location loc-to - location driver - driver)
PRE:	(at truck loc-from),(driving driver truck), (path loc-from location loc-to location)
ADD:	(at truck loc-to),(empty truck)
DEL:	(at truck loc-from)
ACTION	walk(driver - driver loc-from - location loc-to - location)
PRE:	(at driver loc-from), (path loc-from loc-to)
ADD:	(at driver loc-to)
DEL:	(at driver loc-from)

mation $\theta = 5\%$. As we can see, as the training set increases its size, the error rate E_c decreases while the redundancy rate E_r increases. The former effect is due to the fact that the more the training error, the more likely a precondition is explained by a number of plan examples. The increase in E_r can be explained by the fact that as the number of training plans increases, the ratio of actions to goal conditions also increases.

Partial Information In Figure 2(Figure 3),they are trained on the probability threshold $\theta = 80\%$. How does the partial state information affect the complexity of the algorithm? When more intermediate state are known in the plans, the number of clauses becomes larger, therefore the CPU time increases. But it makes the solution better, because every intermediate predicate is a candidate for a precondition or a effect with hight weight.

Probability Threshold In Figure 4(Figure 5),they are trained on the threshold of partial information $\theta = 0\%$. Note that the CPU time is needed by the whole program (including MAXSAT's running time). As we can see, as the probability threshold θ increases, the number of clauses decreases, the CPU time decreases. There are two reasons to explain how the two error rates change. One reason is that we give the higher heuristic weight to the predicates in the add list than in the precondition. The other reason is that the number of plan constraints decreases when the probability threshold θ increases, and the ARMS system produces an action model according to the action, plan and partial information constraints. Once this happens, the action model becomes simple when the number of plan constraints decreases. The number of the predicates in the precondition in the action models deceases, but the number of the predicates in the add list increases. Therefore the error rate E_c decreases and the redundancy rate E_r increases while the probability threshold is increased.

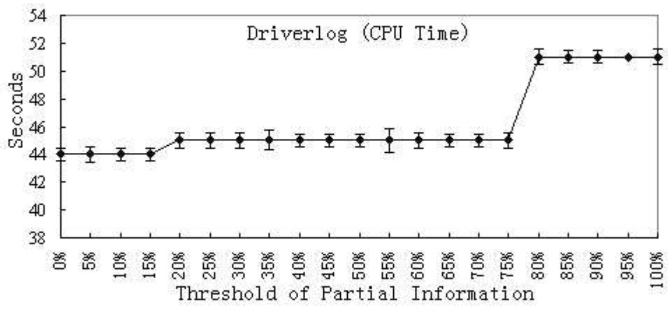

Figure 2: CPU Training Time (Partial Information)

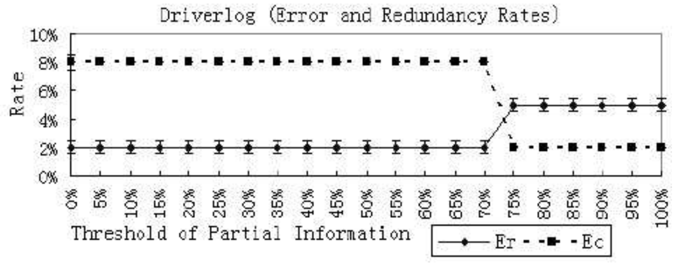

Figure 3: Error and Redundancy Rates(Partial Information)

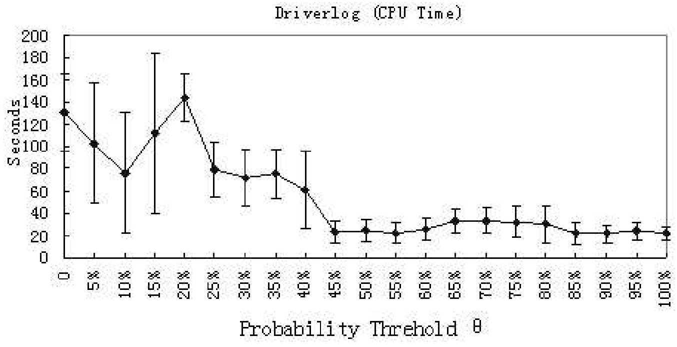

Figure 4: CPU Training Time (Probability Threshold)

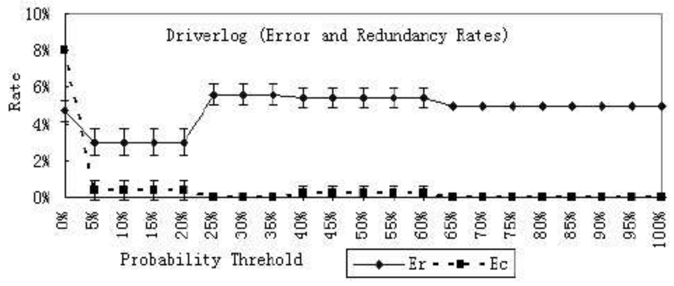

Figure 5: Error and Redundancy Rates(Probability Threshold)

Conclusions

ARMS is a system for automatically discovering action models from a set of observed plans where the intermediates states are either unknown or only partially known. ARMS operates in two phases, where it first applies a frequent set mining algorithm to find the frequent subsets of plans that need be explained first and then applies a SAT algorithm for finding a consistent assignment of preconditions and effects. We also introduce how to evaluate action model.

Our work can be extended in several directions. First, we should add other error measures to evaluate the quality of the learned models. Second, we wish to explore the application of ARMS *iteratively* on sets of actions in a collection of observed plans in the order of decreasing support measure. Finally, the determinate model we considered in ARMS is still too simple to model many real world situations. We wish to extend it to the probabilistic planning domain.

References

Agrawal, R., and Srikant, R. 1994. Fast algorithms for mining association rules. In *Proc. 20th VLDB*, 487–499. Morgan Kaufmann.

Benson, S. 1995. Inductive learning of reactive action models. In *International Conference on Machine Learning*, 47–54. San Fracisco, CA: Morgan Kauffman.

Blythe, J.; Kim, J.; Ramachandran, S.; and Gil, Y. 2001. An integrated environment for knowledge acquisition. In *Intelligent User Interfaces*, 13–20.

Borchers, B., and Furman, J. 1999. A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization* 2(4):299–306.

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

Fink, E., and Yang, Q. 1997. Automatically selecting and using primary effects in planning: Theory and experiments. *Artificial Intelligence* 89(1-2):285–315.

Fox, M., and Long, D. 2003. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)* 20:61–124.

Garland, A., and Lesh, N. 2002. Plan evaluation with incomplete action descriptions. In *Proceedings of the Eighteenth National Conference on AI (AAAI 2002)*, 461 – 467. Menlo Park, California: AAAI Press.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. Pddl—the planning domain definition language.

Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Eleventh Intl Conf on Machine Learning*, 87–95.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on AI (AAAI 96)*, 1194–1201. Menlo Park, California: AAAI Press.

McCluskey, T. L.; Liu, D.; and Simpson, R. 2003. Gipo ii: Htn planning in a tool-supported knowledge engineering environment. In *The International Conference on Automated Planning and Scheduling (ICAPS03)*.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

Oates, T., and Cohen, P. R. 1996. Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the Thirteenth National Conference on AI (AAAI 96)*, 865–868. Menlo Park, California: AAAI Press.

Pednault, E. 1986. Formulating multiagent, dynamic-world problems in the classical planning framework. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, 47 – 82. Morgan Kaufmann.

Sablon, G., and Bruynooghe, M. 1994. Using the event calculus to intetgrate planning and learning in an intelligent autonomous agent. In *Current Trends in AI Planning*, 254 – 265. IOS Press.

Shen, W. 1994. *Autonomous Learning from the Environment*. Computer Science Press, W.H. Freeman and Company.

Wang, X. 1995. Learning by observation and practice: An incremental approach for planning operator acquisition. In *International Conference on Machine Learning*, 549–557.

Winner, E., and Veloso, M. 2002. Analyzing plans with conditional effects. In *Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS-2002)*.

Yang, Q.; Wu, K.; and Jiang, Y. 2005. Learning action models from plan examples with incomplete knowledge. In *The International Conference on Automated Planning and Scheduling (ICAPS05)*.

Zhang, H. 1997. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, 272–275.