The Knowledge Engineering Review, Vol. 22:2, 135–152. © 2007, Cambridge University Press
doi:10.1017/S0269888907001087 Printed in the United Kingdom

ARMS: an automatic knowledge engineering tool for learning action models for AI planning

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, Sun Yat-Sen University (Zhongshan University), Guangzhou, China*
e-mail: kanghengwu@hotmail.com, qyang@cs.ust.hk, lncsri05@zsu.edu.cn

Abstract

We present an action model learning system known as ARMS (Action-Relation Modelling System) for automatically discovering action models from a set of successfully observed plans. Current artificial intelligence (AI) planners show impressive performance in many real world and artificial domains, but they all require the definition of an action model. ARMS is aimed at automatically learning action models from observed example plans, where each example plan is a sequence of action traces. These action models can then be used by the human editors to refine. The expectation is that this system will lessen the burden of the human editors in designing action models from scratch. In this paper, we describe the ARMS in detail. 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 MAXSAT solver. Furthermore, we show empirical evidence that ARMS can indeed learn a good approximation of the finally action models effectively.

1 Introduction

Artificial Intelligence (AI) planning algorithms, such as GraphPlan (Blum & Furst, 1997), TLPlan (Bacchus & Kabanza, 2000) and HTN planning systems (Nau *et al*., 2003), have been shown to generate sophisticated plans for many applications, such as NASA's robotic task planning (Bresina *et al*., 2005), Web services (Pistore *et al*., 2005) and automated manufacturing (Nau *et al*., 2005). All these 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 such as the STRIPS language (Fikes & Nilsson, 1971), ADL (Pednault, 1986) and PDDL (planning domain definition language) (Fox & Long, 2003). However, action models must still be designed by human experts for these planning systems to work.

In the past, the process of designing action models for any domain has been difficult, error-prone and time consuming. Action models are important for AI planning systems. 'No matter how efficient or powerful Planning engines are, they are only as good as the application knowledge that they use. If the planning domain model is flawed, the resulting planning application will be flawed'[1]. Owing to its difficulty and importance, various approaches (Blythe & Kim, 2001; Wang, 1995; Oates & Cohen, 1996; Gil, 1996; Shen, 1994; Sablon & Boulanger, 1994; Benson, 1995) have been proposed to learn action models from plans. A common feature of this kind of work requires the states just before or after each action to be known. However, this requirement is also difficult to satisfy, since in many real world situations we only have the action names and parameters in a sequence, and do not know the states in detail.

[1] http://scom.hud.ac.uk/scomtlm/competition/.

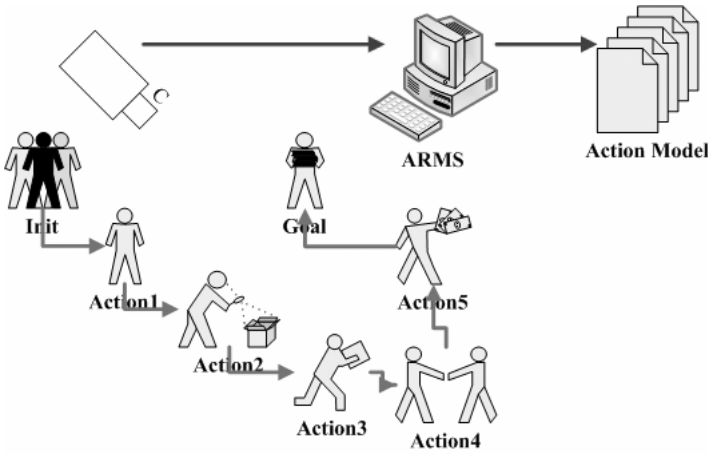


Figure 1 Overall picture of ARMS

We solve this problem with ARMS, which stands for *Action-Relation Modeling System*[2]. ARMS can learn a simple action model from observed plans and verify the action model with an external planning system. The input of ARMS is a set of observed plans, each of which consists of a sequence of instance actions, initial states and goal states. ARMS can automatically generate a simple logical model, irrespective of whether there be some or no intermediate states in the plan trace. 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 *et al.*, 2003).

Figure 1 shows an overall picture of the system, which is motivated by the fact that in many cases, it is necessary to guess a domain model from some agent's goal-directed behaviour in the absence of intermediate-state information. It is easy to observe and record the sequences of executed actions to achieve some goals, but to guess the domain-specific action model is much harder.

For example, we can automatically detect pedestrians or workers completing a task in video sequences and annotate their actions. However, it is hard for a human annotator to explain what they are doing before and after each action in a sequence. It is thus intriguing to ask whether we could intelligently guess an action model in an application domain if we are given only a set of recorded action occurrences and partial or even no intermediate state information.

ARMS proceeds in three phases (Yang *et al.* 2005). In the first phase, it converts all action instances to their schema forms by their corresponding variable types instead of all constants. Thus, an action *pickup*(*box*35) becomes *pickup*(BOX-TYPE) where BOX-TYPE is a box type defined in the domain. This is done for the whole plan example set.

In the second phase, ARMS finds the frequent predicate–action and action sets from the converted plans that share the same object types. Likewise, frequent proposition–action relations can be obtained where the propositions are obtained from the initial and goal states, and the actions are from the first and the last actions in every plan trace. ARMS takes these frequent sets as an initial heuristic approach to get an action model. As an example, if a plan starts with the box located on the floor, with a fact *On*(*box*35, *floor*12) just before the *pickup* action, and if this pair occurs often, we have a frequent pair {*On*(BOX-TYPE, FLOOR-TYPE), *pickup*(BOX-TYPE)} together with its support value (frequency value).

In the last phase, ARMS transforms the frequent sets into a weighted SAT representation, which can be solved by a MAXSAT solver. The frequency values are used as weights to generate an action model using the solution of the MAXSAT problem.

[2] http://www.cs.ust.hk/~khwu/arms.html.

If the set of the training observed plans is very large, the corresponding MAXSAT representation is likely to be too complex to be solved with efficiency and speed. In response, ARMS uses a heuristic method to reduce the number of the MAXSAT clauses. ARMS uses a cross-validation method for evaluating the learned action model against different experimental parameters, such as the threshold of the frequent sets. ARMS also measures the correctness of the learned action model with the error and redundancy rates.

The main contributions of ARMS are as follows:

1. ARMS is the first system that can learn automatically from sequences of 'bare' actions and output a predicted action model; we demonstrate through ARMS that it is feasible to do this learning.
2. Our experiments show that for some domains from the past planning competitions, we can learn a good approximate action model efficiently.

The rest of the paper is organized as follows. In Section 2, we discuss related work in more details. In Section 3, the problem of learning action models from observed plan examples is defined. In Section 4, we describe the architecture of ARMS. In Section 5, we show a simple example and examples of the system interface. In Section 6, we develop a cross-validation evaluation strategy for our learned action models and test our algorithm in several different domains. Finally, we draw a conclusion and discuss the future work.

2 Related work

The problem of learning action descriptions is an essential and notoriously challenging aspect in AI planning. Several researchers have studied this problem in the past decades. We discuss related work in three main areas: (a) in classical learning action descriptions, since our approach is another method of learning action models; (b) in general tools to create planning domains since our approach serves as an add-on component for them; and (c) in solving SAT problems, since ARMS transforms relations into constraints and uses a MAXSAT solver to produce an action model.

2.1 Classical learning action descriptions

An action model can be learned using the knowledge of intermediate states in the sequence of action traces from the studies of Wang (1995), Oates & Cohen (1996) and Gill (1994). These intermediate states provide information for which predicates may exist in the preconditions and effect of an action. The learned action model can be improved and revised after the learning process.

In Wang (1995), a STRIPS model is constructed by computing the most specific condition consistent with the observed examples. The inputs to their learning system, which is known as OBSERVER, are the description language for the domain, the plans and the problems to allow learning-by-doing operator refinements. Given these inputs, OBSERVER automatically acquires the preconditions and effects (including conditional effects and preconditions) of the operators.

In Oates & Cohen (1996), a domain model for an agent situated in a complex environment can be learned. Their approach assumes that a situated agent has knowledge of the types of actions that it can take, but initially knows nothing of the contexts in which an action produces change in the environment, or what that change is likely to be. But their algorithm, MSDD, needs a history of state descriptions observed by an agent as input.

In Shen (1994) a decision tree is constructed from examples to learn preconditions. However, these works require them to be an incomplete action model as input, and learning can be done only when the intermediate states can be observed.

In Sablon & Boulanger (1994) an inductive logic programming (ILP) approach is adopted to learn action models. ILP can learn well when the positive and negative examples of states before all target actions are given. However, in our problem, only a sequence of 'bare' action names are provided in each example with only the initial and goal conditions known.

The DISTILL system (Winner & Veloso, 2002) is designed to learn program-like plan templates from example plans and shares similar motivation with our work. The motivation of DISTILL is similar to ours, where the aim is to automatically acquire plan templates from large number of example plans. In comparison, we are more focused on learning the action models, where planning will still be delegated to a planner for new problems. In Winner & Veloso (2002), the focus is more on how to distill the plans as a substitute for the planner themselves. Thus, our works are complementary to each other.

Also related to our work is Andrew & Neal (2002), who introduced the idea of evaluating a plan's partial success even when the action model is incomplete. As we will see in the experimental section, this is related to our work on evaluating a learned action model by extending the concepts of cross validation in machine learning literature. Finally, compared with plan recognition (Kautz & Allen, 1986), the target of plan recognition is to infer the goals of a sequence of actions, whereas in our case, we are given the goal and our target is to infer an action model.

2.2 General tools for creating planning domains

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). ARMS can be seen as an add-on component for such knowledge editors which can provide advice for human users, such as GIPO (McCluskey *et al.*, (2003) and ModPlan (Edelkamp & Mehler, 2005).

GIPO is a software tool for the creation of planning domains in its internal object-centered representation language. Since it includes some object-oriented methods in other fields of software engineering such as highly visual development methods, code re-use and fast reliable development, GIPO can help users to create and validate a domain description.

Opmaker is an important component in GIPO (McCluskey *et al.*, 2002). Opmaker can induce a set of parameterized operator descriptions from these examples, removing the need for the user to become involved in complex parameter manipulation within the underlying symbolic logical based language. It can help the domain expert specify the declarative structure of the domain and provide training operator sequences.

ModPlan is an interactive knowledge acquisition and engineering tool for AI planning. It provides automated domain analysis with PDDL learning capabilities. It can provide users with an approximate and incremental solution. Domain experts can use it to solve hard combinatorial problems. It can also provide the intermediate results, which are accessible to improve domain modelling, to tune exploration in generating enhanced plans for domain description inference.

2.3 Solving SAT problems

In propositional logic, the satisfiability (SAT) problem 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; Cheeseman *et al.*, 1991). 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 SAT problem. The weighted MAXSAT solvers assign a weight to each clause and seek an assignment that maximizes the sum of the weights of the satisfied clauses (Borchers & Furman, 1999).

3 Problem statement

3.1 Input data

ARMS will learn a PDDL (level 1, STRIPS) representation of actions from the plans. A planning task is specified in terms of a set of objects, an initial state, a goal formula and a set of action schemata. Initial states, goal states and action schemas are based on a collection of logic symbols. States are sets of logical atoms, that is, instantiated predicates. Every action has a precondition

Table 1 Three plan examples

	Plan1	Plan2	Plan3
Initial	I_1	I_2	I_3
Step1	lift(h1 c0 p1 ds0), drive(t0 dp0 ds0)	lift(h1 c1 c0 ds0)	lift(h2 c1 c0 ds0)
Step2	load(h1 c0 t0 ds0)	load(h1 c1 t0 ds0)	load(h2 c1 t1 ds0)
Step3	drive(t0 ds0 dp0)	lift(h1 c0 p1 ds0)	lift(h2 c0 p2 ds0), drive(t1 ds0 dp1)
Step4	unload(h0 c0 t0 dp0)	load(h1 c0 t0 ds0)	unload(h1 c1 t1 dp1), load(h2 c0 t0 ds0)
Step5	drop (h0 c0 p0 dp0)	drive(t0 ds0 dp0)	drop(h1 c1 p1 dp1), drive(t0 ds0 dp0)
Step6		unload(h0 c1 t0 dp0)	unload(h0 c0 t0 dp0)
Step7		drop(h0 c1 p0 dp0)	drop(h0 c0 p0 dp0)
Step8		unload(h0 c0 t0 dp0)	
Step9		drop(h0 c0 c1 dp0)	
Goal	(on c0 p0)	(on c1 p0)	(on c0 p0)
		(on c0 c1)	(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)

list, which is a list of formulas that must hold in a state for the action to be applicable. An action also has an add list and a delete list that are sets of atoms. A plan is a partial order of actions that, when successively applied to the initial state in consistent order, yields a state that satisfies the goal formula. The input to the `ARMS` is a set of plan examples, where each plan example consists of the following:

1. a list of propositions in the initial state;
2. a sequence of action names and actual parameters (that is, instantiated parameters);
3. a list of propositions that hold in the goal state.

Each plan is successful in that it achieves the goal from the initial state; however, the action model for some or all actions may be incomplete in that some preconditions, add and delete lists may not be completely specified. The plan examples may only provide action names such as drive (t0, ds0, dp0), where t0, ds0 and dp0 are objects, but the plan examples do not give the action's preconditions and effects; these are to be learned by our algorithm. Intuitively, an action model is *good* if it explains as many plan examples as possible in the testing plan examples, and it is the simplest one among all models (we use 'simple' in this paper to mean a small size). This is the principle we wish to adhere to.

For example, consider the `depot` problem from the AI planning competition domains[3]. The `depot` domain involves driving trucks around delivering crates between `depot`s and distributors. As an input, we are given predicates such as (clear ?x:surface) to denote that ?x is clear on top and that ?x is of type 'surface', (at ?x:locatable ?y:place) to denote that ?x (a locatable object) is located at ?y (a place). We are also given a set of plan examples described using action names such as drive (?x:truck ?y:place ?z:place), lift(?x:hoist ?y:crate ?z:surface ?p:place). A complete description of the example is in Table 1, which displays the training examples. Table 2 shows the actions to be learned. An example output from our learning algorithms for the load(?x ?y ?z ?p) action is as follows:

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

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

Table 2 Input domain description for depot planning domain

Domain	Depot
Types	place locatable – object depot distributor – place truck hoist surface – locatable pallet crate – surface
Relations	(at ?x:locatable ?y:place) (on ?x:crate ?y:surface) (in ?x:crate ?y:truck) (lifting ?x:hoist ?y:crate) (available ?x:hoist) (clear ?x:surface)
Actions	drive(?x:truck ?y:place ?z:place) lift(?x:hoist ?y:crate ?z:surface ?p:place) drop(?x:hoist ?y:crate ?z:surface ?p:place) load(?x:hoist ?y:crate ?z:truck ?p:place) unload(?x:hoist ?y:crate ?z:truck ?p:place)

From the examples in Table 1, we wish to learn the preconditions, add and delete lists of all actions to explain these examples. Note that the problem would be easier if we knew the states just before and after each action, such as (drop ?x ?y ?z ?p). However, in our system, we address the situation where we know partially or nothing about the states between the actions; thus we do not know for sure exactly what is true just before each of (load h1 c0 t0 ds0), (drive t0 ds0 dp0), (unload h0 c0 t0 dp0), (drop h0 c0 p0 dp0) as well as the complete state just before the goal state in the first plan in Table 1.

4 The ARMS

Our intention in creating the ARMS is not only simply to develop a tool for learning action models from the example plans, but also to describe the input, output and evaluations of ARMS as a criterion in assisting experts to write domain descriptions.

4.1 Architecture of ARMS

Figure 2 represents the architecture of ARMS.

4.1.1 Data preprocessor

A planning task consists of a set of instance objects, a set of instantiated relations at initial state, a goal formula and a set of instance actions. The instance actions are in partial order, called a plan. It can be applied to the initial state in order and produce a state which satisfies the goal formula. ARMS first converts all action instances to their schema forms by replacing all constants by their corresponding variable types. That is, ARMS generalizes all subplans by substituting all occurrences of an instantiated object in actions with the same variable parameter. This enables us to discover many repeated occurrences of action schema forms in the example plans.

Example 1 *Consider the first plan in Table 1. Every instance action will be converted into its action schema with variable parameters in Table 3. Their relevant parameters are also recorded as connectors, which are used to guess the relations of different actions. If two actions' instance parameters are the same, their variable parameters can be combined into a relevant parameter. Because the instance parameters {h1, c0, ds0} are shared between lift(h1 c0 p1 ds0) and load(h1 c0 t0 ds0), the relevant parameter is {?x1=?x2, ?y1=?y2, ?p1=?p2 } between lift(?x1:hoist ?y1:crate ?z1:surface ?p1:place) and load(?x2:hoist ?y2:crate ?z2:truck ?p2:place). Because the instance parameter*

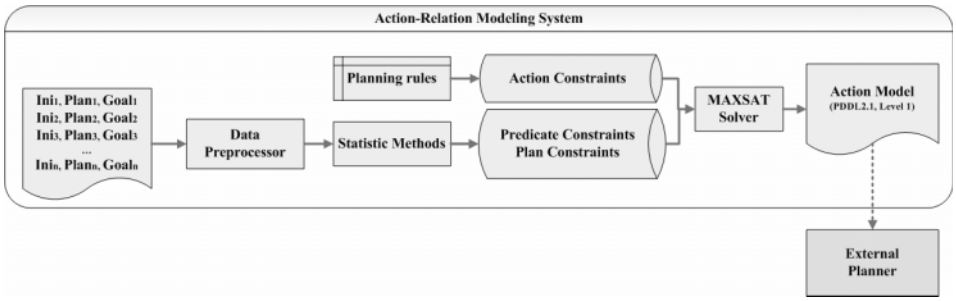


Figure 2 Architecture of ARMS

Table 3 Data preprocessor

Plan1 Initial	Instance action (on c0 p1), etc.	Action schema (on ?x:crate ?y:surface), etc.
1	lift(h1 c0 p1 ds0) drive(t0 dp0 ds0)	lift(?x:hoist ?y:crate ?z:surface ?p:place) drive(?x:truck ?y:place ?z:place)
2	load(h1 c0 t0 ds0)	load(?x:hoist ?y:crate ?z:truck ?p:place)
3	drive(t0 ds0 dp0)	drive(?x:truck ?y:place ?z:place)
4	unload(h0 c0 t0 dp0)	unload(?x:hoist ?y:crate ?z:truck ?p:place)
5	drop (h0 c0 p0 dp0)	drop(?x:hoist ?y:crate ?z:surface ?p:place)
Goal	(on c0 p0), etc.	(on ?x:crate ?y:surface), etc.

{c0, p1} is shared between (on c0 p1) and lift(h1 c0 p1 ds0), the relevant parameter is {?x1=?y2} between (on ?x1:crate ?y1:surface) and lift(?x2:hoist ?y2:crate ?z2:surface ?p2:place).

4.1.2 Statistical method

ARMS then applies a frequent itemset mining algorithm based on the Apriori algorithm (Agrawal & Srikant, 1994), for finding frequent pairs of actions, and pairs of propositions and actions, in the plan examples. As in the data mining literature, we use the term *support* as the percentage of an itemset in a database; if the support is above a user-defined threshold value, we can then call the subset frequent. We do not suffer from the problem of generating too many redundant association rules as in data mining research, since we only apply the Apriori algorithm to find the frequent pairs which are limited in number; these pairs are to be explained by the learning system later. In the experiments, we vary the value of θ to verify the effectiveness of the algorithm. θ equals the number of the action sequence over the maximal number in the set of action sequences, one of which consists of actions. If the number of actions in one action sequence is two, the action sequence is called as the action pair.

$$\theta_i = \frac{\mathrm{NUM}(\mathrm{AS}_i)}{\mathrm{MAX}_{i \in N}(\mathrm{AS}_i)} (\mathrm{AS} : \text{action sequence}).$$

When considering subsequences of actions from example plans, we consider only those sequences whose supports are over θ.

Example 2 *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, c0, ds0 }) from Plan2 can be generalized to lift(?x1 ?y1 ?z1 ?p1), load(?x2 ?y2 ?z2 ?p2) (labelled with {?x1=?x2, ?y1=?y2, ?p1=?p2}). The connector {?x1=?x2, ?y1=?y2, ?p1=?p2} representing the three parameters ?x1, ?y1 and ?p1 in action lift(?x1 ?y1 ?z1 ?p1) are the same as the three parameters ?x2, ?y2 and ?p2 in action load(?x2 ?y2 ?z2 ?p2), respectively. Thus, the support for lift(?x1 ?y1 ?z1 ?p1), load(?x2 ?y2 ?z2 ?p2) with the parameter label {?x1=?x2, ?y1=?y2, ?p1=?p2} is at least two. Table 4 shows all plan constraints*

Table 4 Support of all action sequences

Label	Plan constraints	Number	Support (θ)
{?x1=?x2, ?y1=?y2, ?p1=?p2}	Φ(lift, load)	5	1
{?z1=?x2, ?p1=?y2}	Φ(load, drive)	4	4/5
{?x1=?z2, ?z1=?p2}	Φ(drive, unload)	4	4/5
{?x1=?x2, ?y1=?y2, ?p1=?p2}	Φ(unload, drop)	5	1
{?x1=?x2, ?p1=?p2}	Φ(load, lift)	2	2/5
{?x1=?x2, ?p1=?p2}	Φ(drop, unload)	1	1/5
{?x1=?z2, ?z1=?p2}	Φ(drive, load)	1	1/5
{?y1=?p2}	Φ(drive, load)	1	1/5
{?p1=?p2=?y3}	Φ(lift, load, drive)	4	4/5
{?z1=?x2=?z3}	Φ(load, drive, unload)	4	4/5
{?z1=?p2=?p3}	Φ(drive, unload, drop),	4	4/5
{?p1=?p2=?p3=?y4}	Φ(load, lift, load, drive)	2	2/5
{?z1=?p2=?p3=?p4}	Φ(drive, unload, drop, unload)	1	1/5

along with their support values. In Table 4, Number means the frequent times of the constraints which appear in Table 1, and Support means θ which equals to the number of the constraint over the maximal number in the constraints.

4.1.3 Planning rule

Planning rules represent the requirement of individual actions. Planning rule is given by the domain experts, who are the planning experts. These rules are heuristics to make the learning problem tractable and are reasonable for action definitions, but not for special cases. As an example, a predicate cannot appear in both the add list and del list of an action.

4.1.4 Action constraint

Every action constraint represents a planning rule and receives a constant weight. We define a relation r to be *relevant* to an action a 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 kinds:

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, 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.

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

Constraint A.2. In the real world, actions may cause a physical change in the environment. Thus, ARMS requires that the preconditions, delete and add lists of all actions (except the initial state action) are non-empty. In future, ARMS will consider how to handle the cases where the precondition list is empty.

$$pre_i \neq \phi \wedge add_i \neq \phi \wedge del_i \neq \phi$$

Constraint A.3. 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.

$$pre_i \cap add_i = \phi$$

Constraint A.4. If an action's delete list includes a predicate, it is assumed that this predicate is needed by an instance of the action, that is, an action's precondition includes the predicate. For every action, ARMS requires that the delete list is a subset of the precondition list.

$$del_i \subseteq pre_i.$$

Example 3 *As an example, consider action (lift ?x:hoist ?y:crate ?z:surface ?p:place) in the* `depot` *domain (Table 1). From the draft domain description, the possible predicates of action (lift ?x ?y ?z ?p) are (at ?x ?p), (available ?x), (lifting ?x ?y), (at ?y ?p), (on ?y ?z), (clear ?y), (at ?z ?p) and (clear ?z). And the precondition list* $\texttt{pre}_{goal}$ *of action goal are (on ?c:crate ?p: surface). Suppose that primary effect is (lifting ?x ?y). From this action, the SAT clauses about action (lift ?x ?y ?z ?p) are as follows*:

1. *A possible precondition list includes all of the possible predicates that are joined by a disjunction. From constraint A.1, 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).*
2. *Constraint A.2 can be encoded as the conjunction of the following clauses*:

$$\{ \quad (at\ x\ p) \in \mathrm{pre}_i \vee (available\ x) \in \mathrm{pre}_i \vee (lifting\ x\ y) \in \mathrm{pre}_i \vee (at\ y\ p) \in \mathrm{pre}_i \vee (in\ y\ z) \in \mathrm{pre}_i \vee (clear\ y) \in \mathrm{pre}_i \vee (at\ z\ p) \in \mathrm{pre}_i$$

$$\{ \quad (lifting\ x\ y) \in add_i \vee (at\ y\ p) \in add_i \vee (in\ y\ z) \in add_i \vee (clear\ y) \in add_i \vee (at\ z\ p) \in add_i$$

$$\{ \quad (lifting\ x\ y) \in del_i \vee (at\ y\ p) \in del_i \vee (in\ y\ z) \in del_i \vee (clear\ y) \in del_i \vee (at\ z\ p) \in del_i$$

3. *Constraint A.3 can be encoded as the conjunction of the following clauses*:

$$\{ \quad (lifting\ x\ y) \in add_i \Rightarrow (lifting\ x\ y) \notin pre_i$$

$$\{ \quad (at\ y\ p) \in add_i \Rightarrow (at\ y\ p) \notin pre_i$$

$$\{ \quad (in\ y\ z) \in add_i \Rightarrow (in\ y\ z) \notin pre_i$$

$$\{ \quad (clear\ y) \in add_i \Rightarrow (clear\ y) \notin pre_i$$

$$\{ \quad (at\ z\ p) \in add_i \Rightarrow (at\ z\ p) \notin pre_i$$

$$\{ \quad (lifting\ x\ y) \in pre_i \Rightarrow (lifting\ x\ y) \notin add_i$$

$$\{ \quad (at\ y\ p) \in pre_i \Rightarrow (at\ y\ p) \notin add_i$$

$$\{ \quad (in\ y\ z) \in pre_i \Rightarrow (in\ y\ z) \notin add_i$$

$$\{ \quad (clear\ y) \in pre_i \Rightarrow (clear\ y) \notin add_i$$

$$\{ \quad (at\ z\ p) \in pre_i \Rightarrow (at\ z\ p) \notin add_i$$

4. *Constraint A.4 can be encoded as follows*:

$$\{ \quad (lifting\ x\ y) \in del_i \Rightarrow (lifting\ x\ y) \in pre_i$$

$$\{ \quad (at\ y\ p) \in del_i \Rightarrow (at\ y\ p) \in pre_i$$

$$\{ \quad (in\ y\ z) \in del_i \Rightarrow (in\ y\ z) \in pre_i$$

$$\{ \quad (clear\ y) \in del_i \Rightarrow (clear\ y) \in pre_i$$

$$\{ \quad (at\ z\ p) \in del_i \Rightarrow (at\ z\ p) \in pre_i$$

4.1.5 Predicate constraint

The predicate constraints are derived directly from the statistical method. `ARMS` locates all frequent predicate-action-schema pairs in the plan examples, where the proposition is in the initial state and the action is the first action, or the proposition is in the goal state and the action is the last action.

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, predicate-action pair (clear c1), (lift h1 c1 c0 ds0) (sharing a parameter {c1}) from Plan2 and predicate-action pair (clear c1), (lift h2 c1 c0 ds0) (sharing a parameter {c1}) from Plan3 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 three. Table 5 shows all predicate constraints along with their support values in the previous example.*

Table 5 Examples of all supported predicate constraints

Label	Information constraints	Number	Support (θ)
{?y}	(clear ?y)$\in$ pre_{lift}	3	1
{?x, ?p}	(at ?x ?p)$\in$ pre_{lift}	3	1
{?x }	(available ?x)$\in$ pre_{lift}	3	1
{?z, ?p}	(at ?z ?p)$\in$ pre_{lift}	3	1
{?y, ?p}	(at ?y ?p)$\in$ pre_{lift}	3	1
{?y, ?z}	(on ?y ?z)$\in$ pre_{lift}	3	1
{?x, ?y}	(at ?x ?y)$\in$ $\text{pre}_{\text{drive}}$	1	1/3
{?y, ?z}	(on ?y ?z)$\in$ add_{drop}	3	1

4.1.6 Plan constraint

A plan constraint represents the relationship among 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 to obtain a set of frequent action pairs $<a_i, a_j>$ and predicate-action triples from the plan examples. For each pair of actions in a frequent sequence of actions, we generate a constraint to encode *one* of the following situations. For every action pair, it must be satisfied with one of the three following 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. The first clause is designed for the event when an action uses the fact that is used by a previous action, that is, two actions share the same fact.

$$\exists p(p \in (pre_i \cap pre_j) \wedge p \notin (del_i))$$

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. The second clause is designed for the event when an action uses the fact that is created by a previous action.

$$\exists p(p \in (add_i \cap pre_j))$$

Constraint P.3. A relevant predicate p that is deleted by the first action a_i is added by a_j. The third 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 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 $<a_i, a_{i+1}, \ldots, a_{i+n}>$ must be satisfied. $\Phi(a_i, a_{i+1}, \ldots, a_{i+n})$ can be restated as:

$$\Phi(a_i, a_{i+1}) \wedge \Phi(a_{i+1}, a_{i+2}) \wedge \ldots \wedge \Phi(a_{i+n-1}, a_{i+n})$$

Example 5 *Suppose that (lift ?x1:hoist ?y1:crate ?z1:surface ?p1:place), (load ?x2:hoist ?y2:crate ?z2:truck ?p2: place)) is a frequent pair. The relevant parameter is {?x1=?x2,?y1=?y2, ?p1=?p2}.*

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:

1. *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).*
2. *if f(u) (f(u) 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 f(u) is found as one of the following kinds*:
   i *f(u) 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),*
   ii *f(u) is in the precondition list of action (load ?x ?y ?z ?p) and the add list of (lift ?x ?y ?z ?p),*
   iii *f(u) is in the delete list of action (lift ?x ?y ?z ?p) and add list of (load ?x ?y ?z ?p).*

4.1.7 MAXSAT solver

We build a weighted MAXSAT representation from the three constraints (action constraints, predicate constraints and plan constraints), and solve it using a weighted MAXSAT solver[4]. In solving a weighted MAXSAT problem, 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. *Action constraints*. Every action constraint receives a constant weight, which is called as constant assignment. The constant assignment is empirically determined too, but they are generally higher than the predicate constraints.
2. *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-action pair is higher than the probability threshold θ, the corresponding constraint receives a weight value according 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).
3. *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 of 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.

4.1.8 Action model

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 favour a model that is *correct* and *simple* among all models. Since ARMS gets the solution with the heuristic weight, the model is simple. A plan is said to be *correct* if (a) all actions' preconditions hold in the state just before that action and (b) all goal propositions hold after the last action.

When the probability threshold is set to zero, the solution to a SAT problem, which is formulated using the predicate, action and plan constraints, corresponds to a correct action model. Because the plan constraint is used to explain why an action appears in a plan, each action should be explained when the probability threshold is set to zero. 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. Thus, the plan must be correct.

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

4.1.9 Complexity

Since 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 relations in the preconditions (effects) in one action. 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 $O(n * L * C)$.

In addition, note that owing to the local-search nature of the ARMS algorithm, we cannot guarantee the minimality of the action model theoretically. However, since MAXSAT generates a solution for a SAT problem parsimoniously, the action model generated is generally small. In the next section, we empirically verify the quality of the model.

5 Experimental results

In order to evaluate ARMS, we got the observed plans from the planning domains of International Planning Competition 2002[5], and the plans were generated using the *MIPS* planner[6]. In each domain, we first generated 200 plan examples. We then applied a fivefold cross validation, where we select *160* plan examples as the training set from fourfolds, and use the remaining fold with *40* separate plan examples as the test set. The average length of these plans is *30*. We ran all our experiments on a personal computer with a 768 M memory and a Pentium processor 3.0 GHz CPU using the Linux operating system.

We have done tests on five domains, which are described in Table 6.

5.1 Error rate and redundancy rate

Although it is important to guarantee the correctness of a learned action model on the training examples, ARMS uses a greedy algorithm by explaining only the action pairs 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 possibility that some preconditions remain unexplained.

We define *errors* of an action model M in an example plan P. The error rate E_e 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, E_r is a redundancy rate calculated as the proportion of predicates in an actions's add list that do not establish any preconditions of any action in a later part of the plan. This error is used to measure the degree of redundancy created by the action model in the test dataset.

5.2 Probability threshold

In Figure 3, the errors E_e and E_r on the test data of the depot and driverlog domain is shown. The x-axis represents the probability threshold θ and the y-axis represents the error rate in the range ($0 \leq \theta \leq 10\%$). There are two reasons to explain how the two error rates change. (a) We give the higher heuristic weight to the predicates in the add list than in the precondition. (b) The number of plan constraints decreases when the probability threshold θ increases, and the ARMS produces an action model according to the predicate, action and plan 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 decreases, but the number

[5] http://planning.cis.strath.ac.uk/competition/.
[6] http://www.informatik.uni-freiburg.de/~mmips/.

Table 6 Five domains results

Domain name	Probability threshold (θ) (%)	E_e (%)	E_r (%)	CPU time (Sec.)
Depots	10	19	11	7
Driverlog	10	5	4	21
Zenotravel	10	0	9	7
Rover	60	68	7	224
Satellite	60	47	47	388

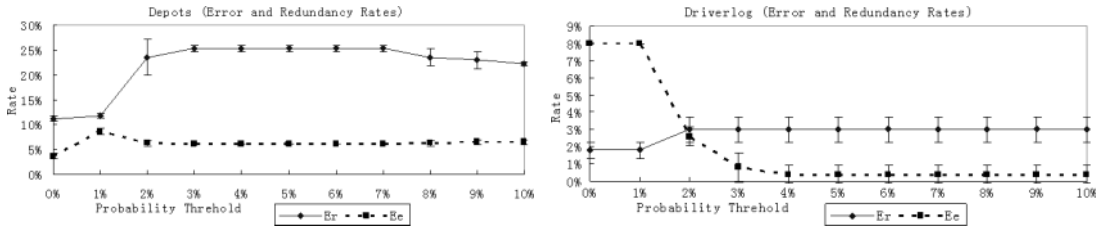


Figure 3 Error and redundancy rates

of the predicates in the add list increases. Therefore the error rate E_e decreases and the redundancy rate E_r increases while the probability threshold is increased.

5.3 Number of clauses

The number of clauses is listed in Figure 4. The x-axis represents the probability threshold θ and the y-axis represents the number of clauses which are encoded into a MAXSAT in the range of ($0 \leq \theta \leq 10\%$). Because the heuristic information about initial states is important, it will be better that the actions of all plans examples are learned in a left-to-right sweep if we assume the plans begin on the left-hand-side and end on the right-hand-side, without skipping any incomplete actions in between. Therefore, `ARMS` 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. Since there are five actions in the `depot` domain, the MAXSAT solver was run five times. Since there are six actions in the driverlog domain, the MAXSAT solver was run six times. From the figure, we can see that the number of clauses is dramatically reduced by a slight increase of the minimum support value in the frequent-set mining algorithm.

5.4 CPU time

The CPU time for training on each training run is listed in Figure 5. The x-axis represents the probability threshold θ and the y-axis represents the CPU time which is needed by the whole program (including MAXSAT's running time) in the range ($0 \leq \theta \leq 10\%$). We can see that as the minimum support θ increases, again the CPU time is shortened in learning the action models. In the driverlog domain, the CPU time is dramatically shortened.

5.5 Comparison between learned action models and ideal action models

we describe the output generated by our SAT solver for the action models learned in the `depot` and driverlog domain. We compare our learned action model with the baseline action model, which is got from the PDDL domain descriptions from IPC 2002. In this example, the action model we use to generate the example plans are called the *ideal* action model (M_i). From the plan examples, we learn a model M_l. In Tables 7 and 8, if a predicate appears in both models, it is shown as normal font. If a predicate appears only in M_i but not in M_l, it is shown as *italic* font. Otherwise, a proposal will only show in M_l and not in M_i, it is in **bold** font. As can be

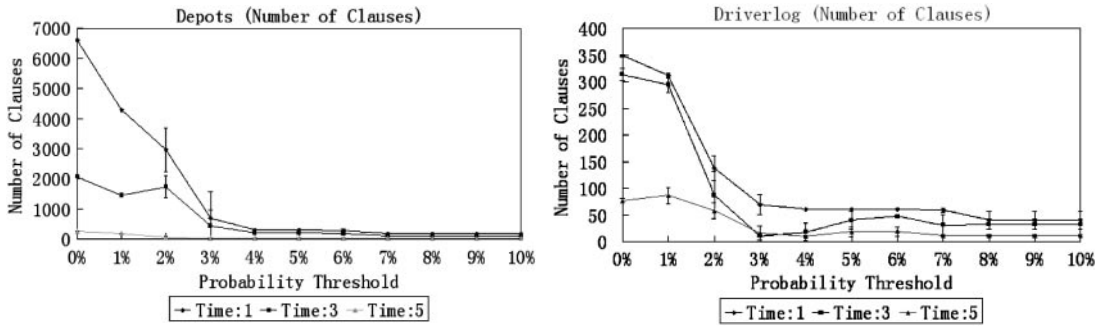


Figure 4 Number of Clauses on Every Time

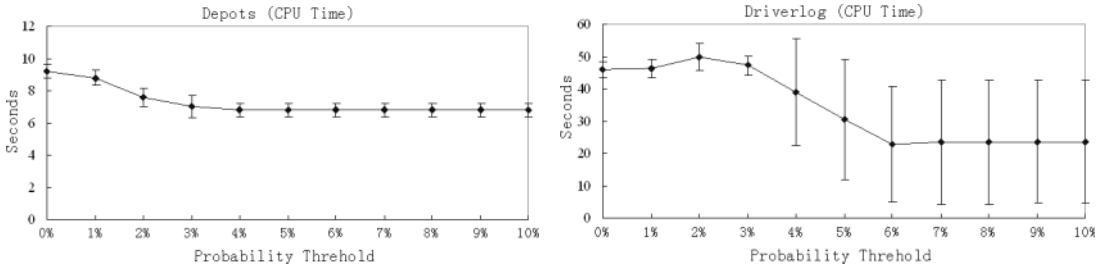


Figure 5 CPU training time

Table 7 The learned action model (Depots Domain, $\theta = 10\%$)

ACTION	drive(?x:truck ?y:place ?z:place)
PRE:	(at ?x ?y)
ADD:	(at ?x ?z)
DEL:	(at ?x ?y)
ACTION	lift(?x:hoist ?y:crate ?z:surface ?p:place)
PRE:	(at ?x ?p),(available ?x),(at ?y ?p),(on ?y ?z), (clear ?y),**(at ?z ?p)**
ADD:	(lifting ?x ?y),(clear ?z)
DEL:	(at ?y ?p),(clear ?y),(available ?x),(on ?y ?z)
ACTION	drop(?x:hoist ?y:crate ?z:surface ?p:place)
PRE:	(at ?x ?p),(at ?z ?p),(clear ?z),(lifting ?x ?y)
ADD:	(available ?x),(clear ?y),(on ?y ?z)
DEL:	(lifting ?x ?y),(clear ?z)
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),**(at ?y ?p), (clear ?y)**
DEL:	(lifting ?x ?y)
ACTION	unload(?x:hoist ?y:crate ?z:truck ?p:place)
PRE:	(at ?x ?p) (at ?z ?p) (available ?x) (in ?y ?z), **(clear ?y)**
ADD:	(lifting ?x ?y)
DEL:	(in ?y ?z),(available ?x),**(clear ?y)**

seen in Tables 7 and 8, in this planning domain, most parts of the learned action model are correct (with respect to the domain given in the AI planning competition).

6 Discussion

If there are partial states in a plan, our algorithm framework can be extended easily to use it. We can treat the partial states as the observed intermediate states which exist in a plan. We can treat

Table 8 The learned action model (Driverlog Domain, $\theta = 10\%$)

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)

them as a new kind of constraints within our algorithm framework. Suppose we observe a proposition p between two actions a_n and a_{n+1}, and p, a_{i1}, …, a_{ik-1} and a_{ik} (a_n) share the same constant parameters, we can present the facts by the following clauses. a_{i1}, …, and a_{ik} appear in the plan's partial order. The action a_{ik} is the action a_n. Therefore, we can get the following two constraints.

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}.

$$p \in (add_{i1} \cup add_{i2} \cup \ldots \cup add_{ik})$$

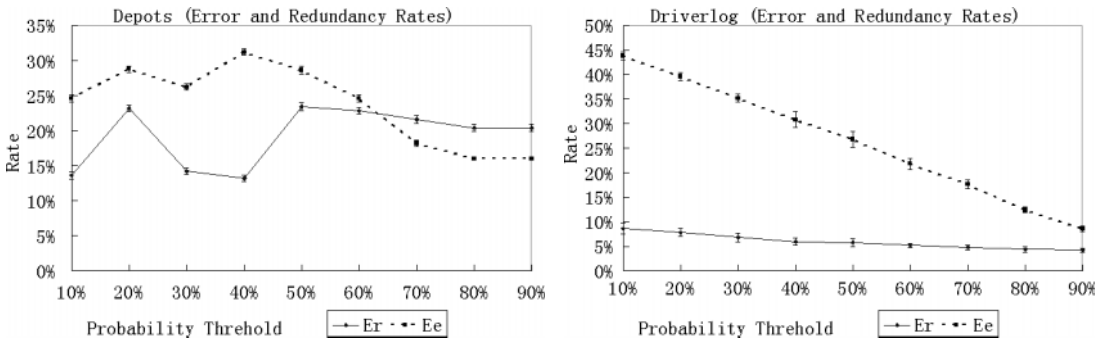
For example, we observe the proposition (at t0 dp0) between two actions drive(t0 ds0 dp0) (Step3) and unload(h0 c0 t0 dp0) (Step4) in Plan1 in Table 1. Because the proposition (at t0 dp0) does not appear in the initial state, we can know at least one action $a(a \in \{\mathrm{lift(h1\ c0\ p1\ ds0)}, \mathrm{drive(t0\ dp0\ ds0)}, \mathrm{load(h1\ c0\ t0\ ds0)}, \mathrm{drive(t0\ ds0\ dp0)}\})$ should generate the proposition (at t0 dp0). Therefore, the proposition (at t0 dp0) is at least in the add list of one action a.

Constraint I.2. The last action a_ik must not delete the predicate p, that is, p must not be selected to be in the del list of a_ik.

$$p \notin del_{ik}$$

For example, we observe the proposition (at t0 dp0) between two actions drive(t0 ds0 dp0) (Step3) and unload(h0 c0 t0 dp0) (Step4) in Plan1 in Table 1. If the action drive(t0 ds0 dp0) delete the proposition (at t0 dp0), we cannot observe the proposition (at t0 dp0) next to the action drive(t0 ds0 dp0). Therefore, the proposition (at t0 dp0) is not in the delete list of the action drive(t0 ds0 dp0).

Every partial information constraint receives a constant weight. The constant assignment is empirically determined too, but they are generally highest in the constraints' weights.



In Figure 6, they are trained on the probability threshold $\theta = 10\%$. How does the partial state information affect the complexity of the algorithm? When more intermediate states are known in the plans, the number of clauses becomes larger, therefore the CPU time increases. It makes the solution better, because every intermediate predicate is a candidate for a precondition or a effect with a high weight.

7 Conclusions and future work

In this paper, we described our ARMS for learning action models from a set of plan examples where the intermediate states are unknown. Our ARMS operates in three basic steps: (a) convert all action instances to their schema forms, (b) find the frequent predicate-action and actions sets from the plan and transform the sets into constraints, and (c) generate a weighted SAT representation and solve it with a MAXSAT solver. Two new error measures are also used to test the learned action models.

ARMS is still under development and can be extended in several directions. First, the PDDL2.1 (level 1, STRIPS) representation of actions from the plans is too simple to be used in real world applications. We wish to extend ARMS to the full PDDL involving both time and many types of resources. We believe that quantification can be learned by considering further compression of the learned action model from a SAT representation. Another direction is to introduce non-deterministic effects into the learned models, which can represent the inherent uncertainty present in the effects of many decisions (Younes & Littman, 2004). We will also explore real-world data from logistics industry and web service composition (Pistore *et al.*, 2005; Wu *et al.*, 2003).

Second, although we have described two new error measures to evaluate the learned model, we still have to find a way to help a human understand both key elements of an individual action and important differences among alternative actions in plan generation. We believe that this can be accomplished with a domain metatheory (Myers, 2005), since the metatheory can provide a semantic framework for guiding the choice of concepts used in summarizing and comparing plans, and specifies important semantic properties of templates, planning variables and instances.

Finally, we wish to explore the application of ARMS on sets of actions in a collection of plans in the order of decreasing support measure. Even when ARMS fails to generate action models from a set of given plans, its execution may result in a partial solution. Although such a partial solution would not explain the complete set of plans, it might solve part of the plans, leaving a smaller and more manageable problem for the domain experts. This allows the most frequent action sets to be explained first, thus causing fast convergence and producing superior action models. In our future work, we wish to carefully test this hypothesis.

Acknowledgement

We are grateful to Roman Bartak, Lee McCluskey and the anonymous referees for useful comments.

References

Agrawal Rakesh and Srikant Ramakrishnan, 1994 Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, pp. 487–499.

Andrew Garland and Neal Lesh, 2002 Plan evaluation with incomplete action descriptions. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence 2002*. AAAI, pp. 461–467.

Bacchus Fahiem and Kabanza Froduald, 2000 Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* **116** 123–191.

Benson Scott, 1995 Inductive learning of reactive action models. In *International Conference on Machine Learning*, pp. 47–54.

Blum Avrim and Furst Merrick, 1997 Fast planning through planning graph analysis. *Artificial Intelligence* **90** 281–300.

Blythe Jim, Kim Jihie, Ramachandran Surya and Gil Yolanda, 2001 An integrated environment for knowledge acquisition. In *Intelligent User Interfaces*. ACM, pp. 13–20.

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

Bresina John, Jonsson Ari, Morris Paul and Rajan Kanna, 2005 Activity planning for the mars exploration rovers. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, pp. 40–49.

Cheeseman Peter, Kanefsky Bob and Taylor William M., 1991 Where the really hard problems are. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, pp. 331–337.

Edelkamp Stefan, and Mehler Tilman, 2005 Knowledge acquisition and knowledge engineering in the modplan workbench. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, pp. 26–33.

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

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

Fox Maria and Long Derek, 2003 PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* **20** 61–124.

Garland Andrew and Lesh Neal, 2002 Plan evaluation with incomplete action descriptions. In *Proceedings of the Eighteenth National Conference on AI (AAAI 2002)*. AAAI, pp. 461–467.

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

Kautz Henry and Selman Bart, 1996 Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence(AAAI)*. AAAI, pp. 1194–1201.

Kautz Henry A. and Allen James F., 1986 Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI)*. AAAI, pp. 32–37.

McCluskey Thomas Leo, Liu D. and Simpson Ron M., 2003 Gipo ii: Htn planning in a tool-supported knowledge engineering environment. In *Proceedings of the International Conference on Automated Planning and Scheduling(ICAPS)*. AAAI, pp. 92–101.

McCluskey Thomas Leo, Richardson, N. and Simpson Ron M., 2002 An Interactive Method for Inducing Operator Descriptions. In *Proceedings of the 6th International Conference on AI Planning and Scheduling (AIPS-2002)*. AAAI.

Moskewicz Matthew W., Madigan Conor F., Zhao Ying, Zhang Lintao and Malik Sharad, 2001 Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (DAC)*. ACM.

Myers Karen L., 2005 Metatheoretic Plan Summarization and Comparison. In *Proceedings of the ICAPS-05 Workshop on Mixed-initiative Planning and Scheduling*. AAAI.

Nau Dana S., Au Tsz-Chiu, Ilghami Okhtay, Kuter Ugur, William Murdock J., Wu Dan and Yaman Fusun, 2003 Shop2: An htn planning system. *Journal of Artificial Intelligence Research* **20** 379–404.

Nau Dana S., Au Tsz-Chiu, Ilghami Okhtay, Kuter Ugur, William Murdock J., Wu Dan and Yaman Fusun, 2005 Applications of shop and shop2. *IEEE Intelligent Systems*, **20**(2) 34–41.

Oates Tim and Cohen Paul R., 1996 Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the Thirteenth National Conference on AI (AAAI 96)*. AAAI, pp. 865–868.

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

Pistore Marco, Traverso Paolo and Bertoli Piergiorgio, 2005 Automated composition of web services by planning in asynchronous domains. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, pp. 2–11.

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

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

Wang Xuemei, 1995 Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning (ICML)*. Morgan Kaufmann, pp. 549–557.

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

Yang Qiang, Wu Kangheng and Jiang Yunfei, 2005 Learning action models from plan examples with incomplete knowledge. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, pp. 241–250.

Younes Haakan L. S. and Littman Michael L., 2004 PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects. In *CMU-CS-04-167, Carnegie Mellon University*.

Wu D., Sirin E., Hendler J., Nau D. and Parsia B., 2003 Automatic web services composition using SHOP2. In *Twelfth International World Wide Web Conference (WWW2003)*. ACM.

Zhang Hantao, 1997 SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE)*. Springer, pp. 272–275.