Learning Action Models from Plan Examples with Incomplete Knowledge

Qiang Yang[1], **Kangheng Wu** [1,2] **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
qyang@cs.ust.hk, khwu@cs.ust.hk and lncsri05@zsu.edu.cn

Abstract

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. In this paper, we develop an algorithm called ARMS for automatically discovering action models from a set of successful plan examples. Unlike the previous work in action-model learning, we do not assume complete knowledge of states in the middle of the example plans; that is, we assume that no intermediate states are given. This requirement is motivated by a variety of applications, including object tracking and plan monitoring where the knowledge about intermediate states is either minimal or unavailable to the observing agent. 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, our ARMS algorithm gathers knowledge on the statistical distribution of frequent sets of actions in the example plans. It then builds a propositional satisfiability (SAT) problem and solves it using a 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) and the PDDL language (Ghallab *et al.*). With these languages, a domain expert sits down and writes a complete set of domain action representation. 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 this difficulty, various approaches (Blythe *et al.* 2001; Wang 1995; Oates & Cohen 1996; Gil 1994; Shen 1994; Sablon & Bruynooghe 1994; Benson 1995) have been explored to learn action models from examples. A common feature of these works is that they require states just before or after each action to be known. Statistical and logical inferences can then be made to learn the actions' preconditions and effects.

In this paper, we take one step ahead in the in direction of learning action models from plan examples with incomplete knowledge. The resultant algorithm is called ARMS , which stands for *Action-Relation Modelling System*. We assume that each example 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 completely unknown to us; that is, between every adjacent pair of actions, the truth of a literal is unknown. Our objective is to reconstruct the literals based on the example plans provided. Suppose that we have several examples as input. From this incomplete knowledge, our system automatically guess a minimal logical action model that can explain most of the plan examples. This action model is not guaranteed to be completely correct, but we hope that it serves as a basis for human designers to build a correct model later on. This paper is about how to build this action model automatically.

As an example, consider the depots problem from the AI Planning PDDL domains (Ghallab *et al.*). As an input, we are given predicates such as $(clear\ x - surface)$ to denote that x is clear on top, $(at\ x - locatable\ y - place)$ to denote that x is located at y, and so on. We are also given a set of plan examples described using action names such as $(drive\ x - truck\ y - place\ z - place)$, $(lift\ x - hoist\ y - crate\ z - surface\ p - place)$, and so on. A complete description of the example is as shown in Table 1, which shows the actions to be learned, and Table 2, which displays the training examples.

From the examples in Table 2, 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 paper, we address the situation where we know little or none 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 2.

In each plan, any proposition such as $(on\ c0\ p0)$ in the goal conditions might be established by the first action,

Table 1: Input Domain Description for Depot

domain	Depot
types	place locatable - object depot distributor - place truck hoist surface - locatable pallet crate - surface
predicates	(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)

the second, or any of the rest. It is this uncertainty that gives difficulty to previous approaches. In contrast from the previous approaches, from these example plans, our algorithm generates a good possible model in which the actions' preconditions and 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:	(lifting x y)
del:	(at y p), (in y z), (available x), (clear y)

The above exemplifies many real world situations. In the area of human and object monitoring and surveillance, cameras and sensors provide only a partial picture of the world in each intermediate state. To encode the actions compactly, it would be nice to have an automatic or semi-automatic method for a good initial guess on the actions seen. Being able to learn the action representations from plan traces can find many applications in location based services in pervasive computing and security. For example, in our test application domain in pervasive computing (Yin, Chai, & Yang 2004), we collect a large sample of a student's movement activities in a university environment using the student's actions to label each movement. We record a student's initial location and status along with the final objective in each plan sequence. However, we have found that it is exceedingly hard to ask the student to explain fully, either from camera snapshots or from mobile transmission signals, what is true of the environmental conditions before and after each intermediate action. The same situation arises from robotic applications where a robot, armed with a number of sensors, might be only able to record its own actions together with some sensory readings, but not able to record other external conditions outside the range of the sensors. This situation corresponds to the case where we know partially the intermediate state information, and we are trying to infer the full action model from this partial information.

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 paper, we take a first step to answer this question by presenting an algorithm.

Table 2: Three plan examples

	Plan1	Plan2	Plan3
Initial state	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 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)

ARMS proceeds in two phases. In phase one of the algorithm, ARMS finds frequent action sets from plans that share a common set of parameters. In addition, ARMS finds some frequent predicate-action pairs with the help of the initial state and the goal state. These predicate-action pairs give us an initial guess on the preconditions, add lists and delete lists of actions in this subset. These action subsets and pairs 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 (Moskewicz *et al.* 2001; Kautz & Selman 1996; Zhang 1997), 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 plan examples, 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 a definition of error 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. The problem statement section defines the problem of learning action models from plan examples. 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).

Problem Statement

In this paper, we will learn a PDDL (level 1, STRIPS) representation of actions. A planning task $\mathcal{P}$ is specified in terms of a set of objects O, an initial state $\mathcal{I}$, a goal formula $\mathcal{G}$, and a set of operator schemata $\mathcal{O}$. $\mathcal{I}$, $\mathcal{G}$, and $\mathcal{O}$ are based on a collection of predicate symbols. States are sets of logical atoms, i.e., instantiated predicates. Every action has a precondition 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 algorithm is a set of plan examples, where each plan example consists of (1) a list of propositions in the initial state, (2) a list of propositions that hold in the goal state, and (3) a sequence of action names and actual parameters (that is, instantiated parameters). 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 a simplest one among all models (we use "simple" in this paper to mean a small size). This is the principle we wish to adhere.

For simplicity, we view the initial state as the first action of a plan, denoted by α_{init}. This action has no preconditions, its add list are the initial-state propositions, and its delete list is an empty set. Similarly, the last action of a plan, α_{goal}, represents the goal of the planning problem. The preconditions of α_{goal} are the goal propositions and its add list and delete list are empty. Note that the initial state (the goal state) and action α_{init} (α_{goal}) are used interchangeably. These two special actions do not need to be learned.

The ARMS Algorithm

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.

Step 7 Output the action model Θ.

The algorithm starts by initializing a set of explained actions and a set of action schemata yet to be explained. Subsequently, it iteratively builds a weighted MAXSAT representation and solve it. Each time a few more actions are explained, and are used to build new initial states. At any intermediate stage, the partially learned action schemata increases in size monotonically. 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 side, 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.

Below, we explain the major steps of the algorithm in detail.

Step 1: Initialize Plans and Variables

A 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 occurrence of an instantiated object in every action instance with the same variable parameter. 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 (see Example 4). Then we initialize Λ be the set of all incomplete action schemata. and the action model Θ by the set of complete action schemata. The action model Θ is empty initially if all actions need be learned.

[1] we use the MAXSAT solver of http://www.nmt.edu/ borchers/maxsat.html

Step 2: Build Action and Plan Constraints

A weighted MAXSAT consists of a set of clauses representing their conjunction. 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. Thus, given a weighted MAXSAT problem, a SAT solver finds a solution by maximally satisfying all the clauses with high weight values. In the ARMS algorithm, we have three kinds of constraints to satisfy, representing three types of clauses in a MAXSAT. They are predicate action, and plan constraints. The predicate constraints are derived from statistics directly (see Step 3 on how to build a weighted MAXSAT).

The second type of clauses in a SAT represents the requirement of individual actions. These constraints are heuristics to make the learning problem tractable and are reasonable for action definitions. 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) In the real world, actions may cause a physical change in the environment. Thus, we require that the preconditions, delete and add lists of all actions (except the initial state action) are non-empty. In future, we will consider how to handle the cases when the precondition list can be empty.
   $pre_i \neq \phi \ \wedge add_i \neq \phi \ \wedge del_i \neq \phi$.
3. (Constraint a.3) Every action's effects cannot negate relations that appear in the precondition list. The intersection of the precondition and add lists of all actions must be empty.
   $pre_i \cap add_i \ = \phi$.
4. (Constraint a.4) If an action's effects do not include a relation, this relation is assumed to be unchanged by an instance of the action. Thus, the unchanged relation cannot appear in the action's precondition either. For every action, we require that the delete list is a subset of the precondition list.
   $del_i \subseteq pre_i$.

Example 1 *As an example, consider the action (load x - hoist y - crate z - truck p - place) in the depot domain (Table 2). From the domain description, 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 constraint 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).*
- *Constraint a.2 can be encoded as the conjunction of the following clauses,*
  - *(at x p)*$\in pre_i \vee$ *(available x)*$\in pre_i \vee$ *(lifting x y)*$\in pre_i \vee$ *(at y p)*$\in pre_i \vee$ *(in y z)*$\in pre_i \vee$ *(clear y)*$\in pre_i \vee$ *(at z p)*
  - *(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)*
  - *(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)*
- *Constraint a.3 can be encoded as the conjunction of the following clauses,*
  - *(lifting x y)*$\in add_i \Rightarrow$ *(lifting x y)*$\notin pre_i$
  - *(at y p)*$\in add_i \Rightarrow$ *(at y p)*$\notin pre_i$
  - *(in y z)*$\in add_i \Rightarrow$ *(in y z)*$\notin pre_i$
  - *(clear y)*$\in add_i \Rightarrow$ *(clear y)*$\notin pre_i$
  - *(at z p)*$\in add_i \Rightarrow$ *(at z p)*$\notin pre_i$
  - *(lifting x y)*$\in pre_i \Rightarrow$ *(lifting x y)*$\notin add_i$
  - *(at y p)*$\in pre_i \Rightarrow$ *(at y p)*$\notin add_i$
  - *(in y z)*$\in pre_i \Rightarrow$ *(in y z)*$\notin add_i$
  - *(clear y)*$\in pre_i \Rightarrow$ *(clear y)*$\notin add_i$
  - *(at z p)*$\in pre_i \Rightarrow$ *(at z p)*$\notin add_i$

  Constraint a.4 can be encoded as follows,
  - *(lifting x y)*$\in del_i \Rightarrow$ *(lifting x y)*$\in pre_i$
  - *(at y p)*$\in del_i \Rightarrow$ *(at y p)*$\in pre_i$
  - *(in y z)*$\in del_i \Rightarrow$ *(in y z)*$\in pre_i$
  - *(clear y)*$\in del_i \Rightarrow$ *(clear y)*$\in pre_i$
  - *(at z p)*$\in del_i \Rightarrow$ *(at z p)*$\in pre_i$

The third type of constraint represents the relationship between actions in a plan to ensure that a plan is correct. Because 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 $< a_i, a_{i+1}, a_{i+2}, ..., a_{i+n} >$ 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 $< a_i, a_{i+1}, a_{i+2}, ..., a_{i+n} >$ 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 del list of a_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, or
- (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.

  Let pre_i, add_i and del_i represent a_i's precondition list, add-list and del-list, respectively. The above plan constraint $\Phi(a_i, a_j)$ can be restated as:
  $\exists p.(p \in (pre_i \cap pre_j) \wedge p1 \notin (del_i)) \vee (p \in (add_i \cap pre_j)) \vee (p \in (del_i \cap add_j))$
- (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 $< a_i, a_{i+1}, a_{i+2}, ..., a_{i+n} >$ must be satisfied.

  Let $\Phi(a_i, a_{i+1})$ represents the constraint of action pair $< a_i, a_{i+1} >$, the above plan constraint $\Phi(a_i, a_{i+1}, a_{i+2}, ..., a_{i+n})$ 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 2 *As an example, consider the depot domain(Table 2). Suppose that ((lift x - hoist y - crate z - surface p - place), (load x - hoist y - crate z - truck p - place), 0) is 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 three 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 3 *As an example, consider the example in Table 2. 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 3 shows all predicate constraints along with their support values in the previous example.*

Table 3: 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. **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.

Example 4 *Consider the example in Table 2. 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 4 shows all plan constraints along with their support values.*

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.

Table 4: 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

Properties of The ARMS Algorithm

Given a set of planning examples, we can find a large number of models, each making all the example plans in the training set correct. However, we would prefer a model that returns the *simple* and *accurate* among all models.

We first formally define the correctness of a plan following the STRIPS model. We consider a plan as a sequence of actions, and a state as a set of atoms.

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.*

Definition 2 *A* simple *action model is one where the total number of predicates assigned to the preconditions, add and delete lists of all actions in the example plans is minimal.*

Definition 3 *An* accurate *action model is one where every relation in the preconditions, add and delete lists of all actions in the example plans is both necessary and correct.*

We prefer good models among all models.

While 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 chance that some preconditions remain unexplained. We define *errors* of an action model M in an example plan P. The error E_b 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_f is an error rate calculated as the proportion of predicates in actions's add list that do 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 error 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. In the next section, we empirically demonstrate that both errors are small in the test plans.

In addition, note that due 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.

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 amounts to applying a planning system to the incomplete action model, and then assess the model's degrees of correctness. This is the approach adopted by (Garland & Lesh 2002), which studies how to select plans that may fail with an aim to minimize risks. In this work, we have available a set of example plans which 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 input 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 our experiments, the training and testing plan sets are generated using the *MIPS* planner (Edelkamp & Helmert 2001) [2]. To generate the initial and goal conditions randomly, we applied the problem generator in International Planning Competition 2002 (IPC 2002) [3]. In each problem 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 lengths of these plans are *50*.

To illustrate, we describe the output generated by our SAT solver for the action model learned in the depot domain. We compare our learned action models with the baseline action models, which are 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 the table below, if a predicate appears in both models, then it is shown as normal font. If a predicate appears only in M_i but not in M_l, then 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

[2]http://www.informatik.uni-freiburg.de/∼mmips/

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

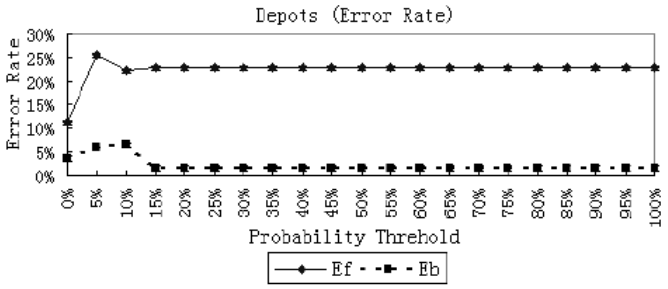

Figure 1: Error Rates in The Depot Domain

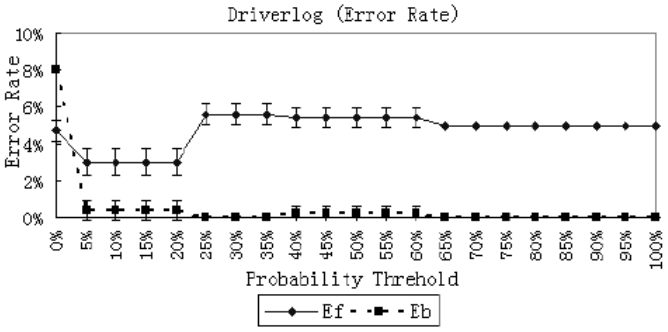

Figure 2: Error Rates in the Driverlog Domain

be seen in Table 5 and Table 6 , in this planning domain, most parts of the learned action model are correct (with respect to the domain given in the AI Planning Competition). Below, we show the experimental results with the two action models.

Error Rate

In Figure 1 and Figure 2, the errors E_b and E_f on the test data of the two domains are shown. E_b (E_f)is used to measure the degree of correctness (redundancy) of the model on the test plans. There are 160 training plans and 40 test plans in each of two domains. The X-axis represents the probability threshold θ and the Y-axis represents the error rate in the range ($0 \leq \theta \leq 100\%$). We can see that as the minimum support increases, the error rate of the learned model quickly converges to a stable point.

In Figure 3 and Figure 4, the errors E_b and E_f on the test data of the two domains are shown. They are trained on the probability threshold $\theta = 80\%$ in each of the domains. These experiments test the effect of increasing the size of the training plan set on the learning quality, which is tested on the fixed test set. The X-axis represents the number of plans and the Y-axis represents the error rate. As we can see, as the training set increases its size, the precondition error E_b decreases while the add-list error E_f 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_f can be explained by the fact that as the number of training plans increases, the ratio of

Table 5: The Learned Action Model(Depots Domain, $\theta = 80\%$)

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)**

Table 6: The Learned Action Model(Driverlog Domain, $\theta = 80\%$)

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)

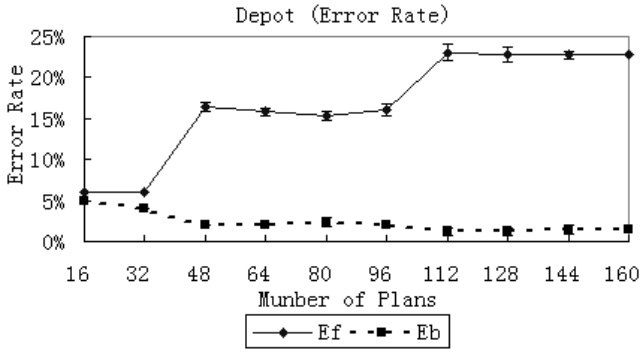

Figure 3: Error Rate of Depot Domain

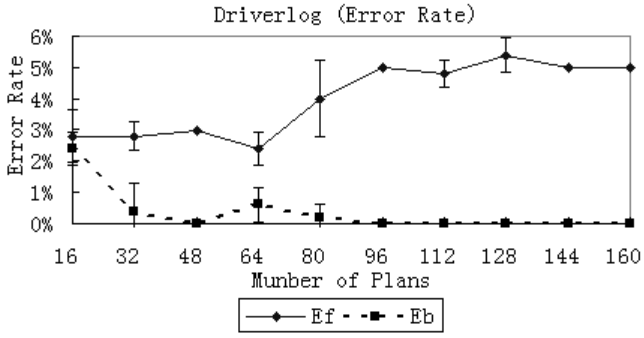

Figure 4: Error Rate of Driverlog Domain

actions to goal conditions will also increase.

Number of Clauses

The number of clauses is listed in Figure 5 and Figure 6. 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 ($0 \leq \theta \leq 100\%$). Since there are five actions in the depot domain, the MAXSAT solver was run five times. From these figures, 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.

CPU Time

We ran all of our experiments on a Pentium III personal computer with 384M memory and 2.0 GHz CPU using the Linux operating system. The CPU time for training time on each training run is listed in Figure 7 and Figure 8. 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 ($0 \leq \theta \leq 100\%$). We can see that as the minimum support θ increases, again the CPU time is dramatically shortened in learning the action models.

The CPU time is listed in Figure 9 and Figure 10. The X-axis represents the number of plans and the Y-axis represents

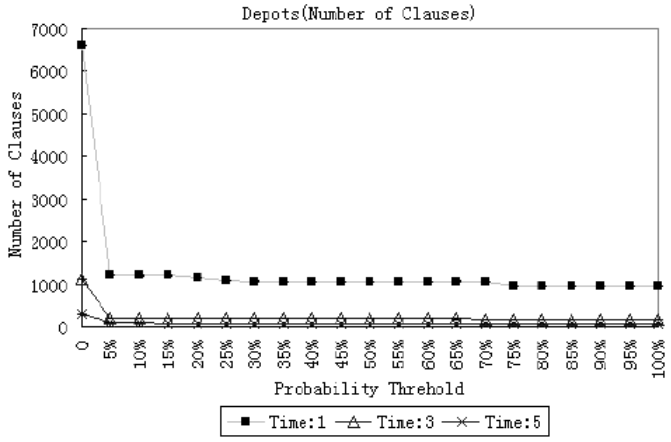

Figure 5: Number of Clauses on Every Time (Depot)

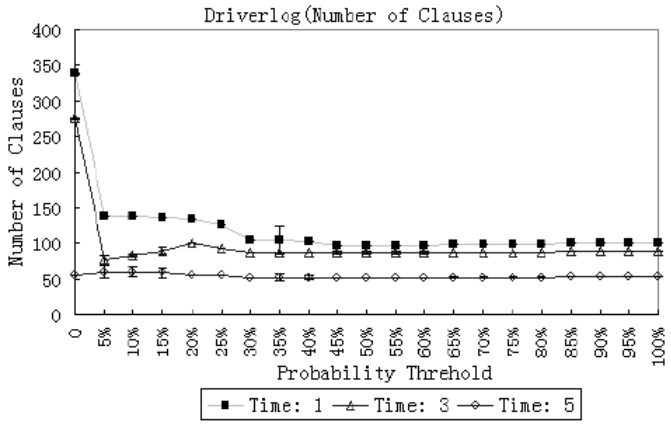

Figure 6: Number of Clauses on Every Time (Driverlog)

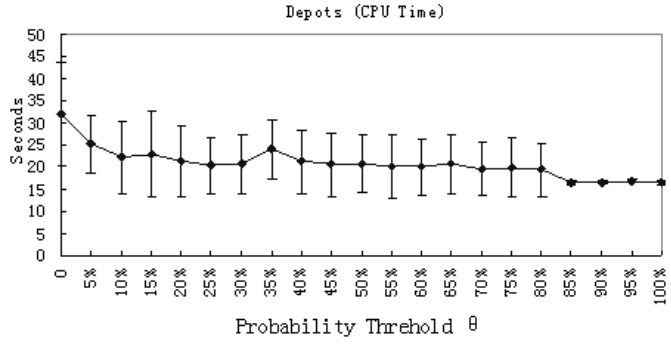

Figure 7: CPU Training Time (Depot)

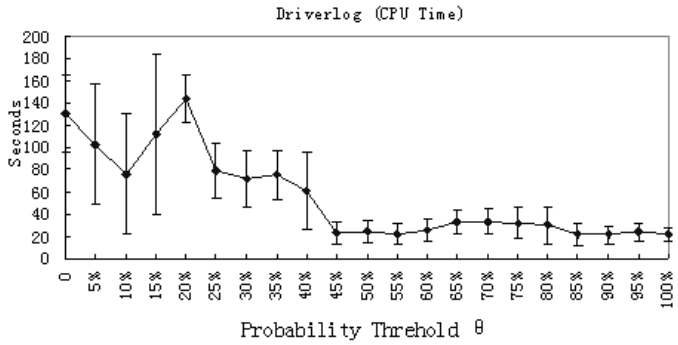

Figure 8: CPU Training Time (Driverlog)

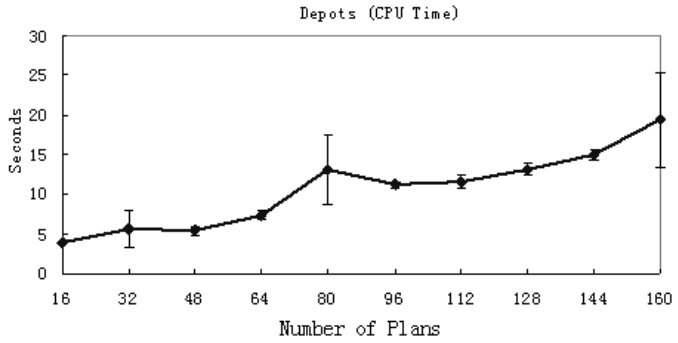

Figure 9: CPU Training Time (Depot)

the CPU time which is needed by the whole program in the range ($0 \leq \theta \leq 100\%$).

From the above figures, we can see that when the probability threshold θ is greater than a certain value, the number of clauses i, CPU time and accuracy are stable.

Although here we only show a couple of domains, these results are encouraging. Even though the training examples consist of a relatively small number of plans, each plan consists of many actions. This provides many action and predicate pairs that in turn gives ample opportunities for learning the action model.

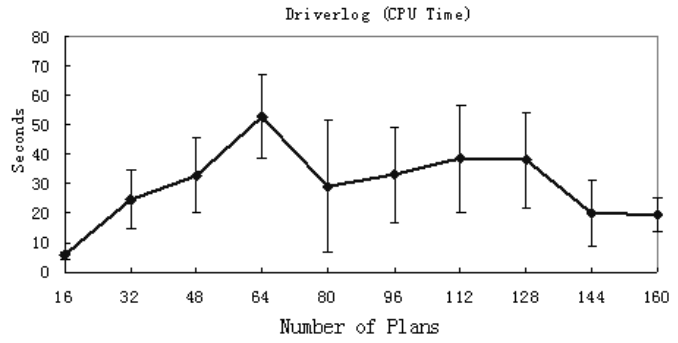

Figure 10: CPU Training Time (Driverlog)

Conclusions and Future Work

In this paper, we have developed an algorithm for automatically discovering action models from a set of plan examples where the intermediates states are unknown. Our ARMS algorithm operates in two phases, in which 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 a test method to evaluate our model.

Our work can be extended in several directions. First, although we have evaluated the system using two new error measures, we still have to find a way to put the learned models to test in plan generation. A major difficulty in doing so is the ability to evaluate the quality of the generated plans using the learned models. We believe that this can be accomplished by interactively repairing the models with the help of a human experts.

Second, the STRIPS language we considered in our paper is still too simple to model many real world situations. We wish to extend to the full PDDL language including quantifiers and time and 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 allow actions to have duration and to consume resources.

Acknowledgement

We wish to thank Jicheng, Zhao for participating in earlier versions of the work. This work is supported by Hong Kong RGC grants HKUST 6180/02E and HKUST 6187/04E. Kangheng Wu was partially supported by the Zhongshan University Kaisi Scholarship.

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.

Edelkamp, S., and Helmert, M. 2001. The model checking integrated planning system (mips). *AI Magazine* 22(3):67–71.

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.

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.

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)*.

Yin, J.; Chai, X. Y.; and Yang, Q. 2004. High-level goal recognition in a wireless lan. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI04)*, 578–583.

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.