Available online at www.sciencedirect.com

ScienceDirect

Artificial Intelligence 171 (2007) 107–143

Artificial Intelligence

www.elsevier.com/locate/artint

Learning action models from plan examples using weighted MAX-SAT

Qiang Yang [a,*], Kangheng Wu [a,b], Yunfei Jiang [b]

[a] *Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clearwater Bay, Kowloon, Hong Kong, China*

[b] *Software Institute, Zhongshan University (Sun Yat-Sen University), Guangzhou, China*

Received 21 October 2005; received in revised form 14 November 2006; accepted 27 November 2006

Available online 25 January 2007

Abstract

AI planning requires the definition of action models using a formal action and plan description language, such as the standard Planning Domain Definition Language (PDDL), as input. However, building action models from scratch is a difficult and time-consuming task, even for experts. In this paper, we develop an algorithm called `ARMS` (action-relation modelling system) for automatically discovering action models from a set of successful observed plans. Unlike the previous work in action-model learning, we do not assume complete knowledge of states in the middle of observed plans. In fact, our approach works when no or partial intermediate states are given. These example plans are obtained by an observation agent who does not know the logical encoding of the actions and the full state information between the actions. In a real world application, the cost is prohibitively high in labelling the training examples by manually annotating every state in a plan example from snapshots of an environment. To learn action models, `ARMS` gathers knowledge on the statistical distribution of frequent sets of actions in the example plans. It then builds a weighted propositional satisfiability (weighted MAX-SAT) problem and solves it using a MAX-SAT solver. We lay the theoretical foundations of the learning problem and evaluate the effectiveness of `ARMS` empirically.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Learning action models; Automated planning; Statistical relational learning

1. Introduction

AI planning systems require the definition of action models, an initial state and a goal. In the past, various action modelling languages have been developed. Some examples are STRIPS [13], ADL [12] and PDDL [14,17]. 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 action models from scratch is a task that is exceedingly difficult and time-consuming even for domain experts. Because of this difficulty, various approaches [4,5,18,34,37,42,43] have been explored to learn action

* Corresponding author.
E-mail address: qyang@cse.ust.hk (Q. Yang).
URL: http://www.cse.ust.hk/~qyang.

models from examples. In knowledge acquisition for planning, many state of the art systems for acquiring action models are based on a procedure in which a computer system interacts with a human expert, as illustrated by Blythe et al. [5] and McCluskey et al. [29]. 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 towards automatically acquiring action models from observed plans in practice. The resultant algorithm is called ARMS, which stands for *Action-Relation Modelling System*. We assume that each observed plan consists of a sequence of action names together with the objects that each action uses. The intermediate states between actions can be partially known; that is, between every adjacent pair of actions, the truth of a literal can be totally unknown. This means that our input can be in the form of action names and associated parameter list with no state information, which is much more practical than previous systems that learn action models. Suppose that we have several observed plans as input. From this incomplete knowledge, our ARMS system automatically guesses the approximately correct and concise action models that can explain most of the observed plans. This action model is not guaranteed to be completely correct, but it can serve to provide important initial guidance for human knowledge editors.

Consider an example input and output of our algorithm in the Depot problem domain from an AI Planning competition [14,17]. As part of the input, we are given relations such as (clear ?x:surface) to denote that ?x is clear on top and that ?x is of type "surface", relation (at ?x:locatable ?y:place) to denote that a locatable object ?x is located at a place ?y. We are also given a set of plan examples consisting of action names along with their parameter list, such as drive(?x:truck ?y:place ?z:place), and then lift(?x:hoist ?y:crate ?z:surface ?p:place). We call the pair consisting of an action name and the associated parameter list an *action signature*; an example of an action signature is drive(?x:truck ?y:place ?z:place). Our objective is to learn an *action model* for each action signature, such that the relations in the preconditions and postconditions are fully specified.

A complete description of the example is shown in Table 1, which lists 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. Once an action is given with the three lists, we say that it has a complete action model. Our goal is to learn an action model for every action in a problem domain in order to "explain" all training examples successfully. An example output from our learning algorithms for the load(?x ?y ?z ?p) action signature is:

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

Table 1
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)

Table 2
Three plan traces as part of the training 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)
State		(lifting h1 c1)	
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)
State	(available h1)		
Step4	unload(h0 c0 t0 dp0)	load(h1 c0 t0 ds0)	unload(h1 c1 t1 dp1), load(h2 c0 t0 ds0)
State	(lifting h0 c0)		
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 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).

We wish to emphasize an important feature of our problem definition, which is the relaxation of observable state information requirements. The learning problem would be easier if we knew the states just before and after each action. However, in reality, what is observed before and after each action may just be partially known. In this paper, we address the situation where we know little about the states surrounding 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. Part of the difficulty in learning action models is due to the uncertainty in assigning state relations to actions. In each plan, any relation such as (on c0 p0) in the goal conditions might be established by the first action, the second, or any of the rest. It is this uncertainty that causes difficulties for many previous approaches that depend on knowing states precisely.

In our methodology, the training plans can be obtained through monitoring devices such as sensors and cameras, or through a sequence of recorded commands through a computer system such as UNIX. These action models can then be revised using interactive systems such as GIPO.

It is thus intriguing to ask whether we can approximate an action model in an application domain if we are given a set of recorded action signatures as well as partial or even no intermediate state information. In this paper, we take a first step towards answering this question by presenting an algorithm known as `ARMS`. `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 relation-action pairs with the help of the initial state and the goal state. These relation-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. We then transform the constraints extracted from the plans into a weighted MAX-SAT representation [25,30,49], solve it and produce action models from the solution of the SAT problem. The process iterates until all actions are modelled. While the action models that we learn in this paper are deterministic in nature, in the future we will extend this framework to learning probabilistic action models to handle uncertainty.

For a complex planning domain that is represented by hundreds of relations and actions, the corresponding weighted MAX-SAT representation is likely to be too large to be solved efficiently as the number of clauses can reach tens of thousands. In response, we design a heuristic method for modelling the actions approximately. We then measure the correctness of the model using a definition of error rates and redundancy rates. For testing the model correctness, we design a cross-validation method for evaluating the learned action models under different experimental conditions such as the different amount of observations and the number of plans.

Our previous work [46] reported on the feasibility of learning action models from action sequences in two planning domains. In this paper, we describe the ARMS system in further detail, add additional constraints that allow partial observations to be made between actions, prove the formal properties of the system, and evaluate ARMS on all STRIPS planning domains from a recent AI Planning Competition.

The rest of the paper is organized as follows. The next section discusses related work in more detail. Section 3 defines the problem of learning action models from plan examples. Section 4 presents the ARMS algorithm. Section 5 presents the experimental results. Section 6 concludes with a discussion of future work.

2. Related work

2.1. Learning from state images

The problem of learning action descriptions is important in AI Planning. As a result, several researchers have studied this problem in the past. In [43], [34] and [18], 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 [43] a STRIPS model is constructed by computing the most specific condition consistent with the observed examples. In [42] a decision tree is constructed from examples in order to learn preconditions. However, these works require there to be incomplete action models as input, and learning can only be done when the intermediate states can be observed. Compared to these works, our work can learn an action model when no state information or only incomplete state information is available.

2.2. Inductive logic programming

In [37] an inductive logic programming (ILP) approach is adopted to learn action models. Similarly, in [4], 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. However, in our problem, only a sequence of "bare" action names are provided in each example with only the initial and goal conditions known. To the best of our knowledge, no work so far has been done to apply ILP to our case. 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.

2.3. Knowledge acquisition in planning

Another related thrust adopts an approach of knowledge acquisition, where the action models are acquired by interacting with a human expert [5]. Our work can be seen as an add-on component for such mixed-initiative knowledge editing systems which can provide advice for human users, such as GIPO [29]. The DISTILL system [44], which is a method used to learn program-like plan templates from example plans, shares similar motivation with our work. The aim is to automatically acquire plan templates from example plans, where the templates can be used for planning directly. Compared to our work, we are more focused on learning the action models, where planning will still be delegated to a planner for new problems. In [44], the focus is more on how to extract plan templates from example plans as a substitute for the planner. The plan templates represent the structural relations among actions. Also related to our work is the work of Garland and Lesh [16], who introduced the idea of evaluating a plan's partial success even when the action models are incomplete. In [27], a programming by demonstration (PBD) problem is solved using a version-space learning algorithm, where the goal is to acquire the normal behavior in terms of repetitive tasks. When a user is found to start a repetitive task of going through a sequence of states, the system can use the learned action sequence to map from initial to goal states directly. This objective relates to our goal in that the observed user traces are used to learn sequences of actions. However, a difference is that with the PBD system [27], the aim is to learn action sequences rather than action models.

Plan traces have also been exploited to learn situation-action rules in [3], where a decision-tree learning algorithm is applied to learn if-then rules that represent a control strategy for an automated agent. From a sequence of sensor readings and control actions taken by human users that represent a complex man-machine operation, such as flying a Boeing 747 jetliner, situation-action rules can be learned to map from sensor readings to actions taken. The condition part of the rules can be considered as a form of preconditions learned to guide the application of the actions in the then part of the rules.

Action-model learning has also been considered for more complex forms of planning as well. CaMeL [21–23] is a candidate-elimination based learning algorithm that acquires the method-preconditions of a hierarchical task network (HTN) planning domain for a given set of task-decomposition schemata [10,45]. These schemata are also known as *methods* [32,33]. The input of CaMel consists of collections of plan traces and derivational trees that are used to derive the plan traces. Their aim is to learn preconditions that control when a decomposition method is applicable. In order to overcome the problem of the lack of training data, in the CeMel system, a stronger form of input is considered as compared to ours, in that the inference process for each plan trace is also taken as part of the input. It would be interesting in future work to consider how to incorporate this additional form of problem-solving knowledge in the learning process, as well as how to extend the learning problem to HTN schemata. We will discuss these issues further in the future work part of the paper.

2.4. Satisfiability problems

A propositional logic formula F can always be transformed into a conjunctive normal form as the conjunction of m clauses, where each clause is the disjunction of a set of literals. A literal is either a variable or the negation of a variable. Given a formula F, we can define a satisfiability problem (SAT) as

> Given a formula F in conjunctive normal form, is there a truth assignment of the logical variables that makes F true?

The satisfiability problem is known to be NP-Complete [15]. There are two classes of solutions to the SAT problem. The first class consists of exact algorithms including logical approaches [8] and integer programming approaches [24]. The second class are heuristics based, such as the GSAT heuristic [38]. These algorithms search for a satisfying truth assignment for a SAT problem.

Recently, extensions of the SAT problems have been studied by several researchers. Related to our problem of learning action models is the weighted MAX-SAT problem which can be stated as

> Given a collection C of m clauses, $C_1, \dots, C_m$ involving n logical variables, with clause weights w_i, find a truth assignment that maximizes the total weight of the satisfied clauses in C.

The weighted MAX-SAT problems are also shown to be NP-hard. Goemans and Williamson [19] have shown that it is feasible to approximate MAX-SAT within a factor of 0.758 from the optimal solution in polynomial time.

The weighted MAX-SAT problems are getting increasing attention in theory and in practice. As a result, several efforts have been made to implement MAX-SAT solvers. In [26,39], an efficient algorithm for solving weighted MAX-SAT problems was presented, and a software tool was made available [11]. Richardson and Domingos [35,36] presented a unified framework for statistical relational learning using a Markov logic network (see a more detailed discussion in Section 2.5). It was suggested that MaxWalkSat can be applied for finding the approximately satisfying assignments of truth values to the ground predicates as a result of logical inference.

In [7], Borchers and Furman describe a two-phase algorithm for solving MAX-SAT and weighted MAX-SAT problems. In the first phase, they use a GSAT heuristic to find a good solution to solving the problem. This solution serves as a seed in generating heuristics for the second phase. In the second phase, they use an enumeration procedure based on the Davis–Putnam–Loveland algorithm [8,28] to find a provably optimal solution. The first heuristic stage improves the performance of the algorithm by obtaining an upper bound on the minimum number of unsatisfied clauses that can be used in pruning branches of the search tree, and use this upper bound to guide further searches.

When compared with an integer programming approach for the problem, they found their algorithm to be better than the integer programming methods for some classes of problems, while the latter is better for other classes. In this paper, we call their method the weighted MAX-SAT solution. The solution is available as an implemented software at http://infohost.nmt.edu/~borchers/maxsat.html.

In this paper, we concern ourselves mainly with the problem of how to convert an action-model learning problem to a weighted MAX-SAT problem. Once this conversion is done, the weighted MAX-SAT problem is then solved using any proven algorithms. Also, in the discussion that follows, we use the term "constraints" and "clauses" interchangeably.

2.5. Markov logic networks

Another related area is Markov logic networks (MLN), which integrates statistical learning in Markov networks and symbolic logic in a powerful and unified learning and inferencing framework [9,35,36]. In a Markov network, there is a set of variables that correspond to the network nodes, $X = (X_i, \dots, i = 1 \dots n)$, where the joint distribution is defined in a log-linear model. Over this network, a first order logic framework is represented by including the logic constructs, that is, constants, predicates, and truth value assignments. In addition, a knowledge base is given that includes a set of first order logic formulae. Two tasks are defined for learning. The first task is to learn the structure of the Markov network, which can be solved using an inductive logic programming approach (ILP). The second task is to learn the parameters of a MLN, the most important component being the weights associated with the features, which can be learned using methods such as conjugate gradient or iterative scaling. Once learned, the MLN can be used to accomplish a number of inferencing tasks.

Once learned, the MLN can be used to perform logical inference. Inference can be in the form of asking a query of a knowledge base, and computation can be done to find truth-value assignments that satisfy all knowledge base formulae. One method to accomplishing this task is to exploit a weighted MAX-SAT solver such as MAXWalkSat, which maximizes the sum of weights of satisfied clauses, or conversely, minimizes the sum of weights of unsatisfied clauses.

The MLN can be used to model and learn the action-model-learning problem described subsequently. There is a direct mapping between the components in action models and the structure and parameters of a MLN. We will return to this correspondence in Section 5.2.

2.6. Relation to SLAF algorithms

Amir et al. [2,40,41] presented a tractable and exact solution to a version of the action-model learning problem using a technique known as Simultaneous Learning and Filtering (SLAF). In this version of the learning problem, the aim is to learn the actions' effects in a partially observable STRIPS domain. The input consists of a finite set of propositional fluents, a set of world states, a finite set of actions as well as a transition relation that maps from states to states by actions. The training examples consist of sequences of actions where partial observations of states are given. The objective is to build a complete explanation of observations by models of actions through a Conjunctive Normal Form (CNF) formula. By tying the possible states of fluents with the effect propositions in the action models, the complexity of the CNF encoding can be controlled to permit exact solutions efficiently in some circumstances.

This work is closely related to ours due to the similar objectives of learning action models. However, the problems being solved by the SLAF techniques are different from ours. In Amir et al.'s work, the aim is to learn action models from state observations; if the state observations were empty, then their technique will not work. In addition, the SLAF methods find exact solutions. In contrast, in our case, we aim to learn *approximate* models from the action sequences with only possibly unknown intermediate states for training.

2.7. PDDL background

PDDL (Planning Domain Definition Language) is designed to formally specify actions and plans in a unified and expressive language inspired by the well-known STRIPS formulation of planning problems [14,17]. There are several main components of PDDL. The actions are represented in a set of schemata, where each action consists of an action name, a typed parameter list, a precondition list and an effect list. The effects can be partitioned into an add list and a

Table 3
A domain description in PDDL

(define	(domain vehicle)
(:requirements	:strips :typing)
(:types	vehicle location fuel-level)
(:predicates	(at ?v:vehicle ?p:location)
	(fuel ?v:vehicle ?f:fuel-level)
	(accessible ?v:vehicle ?p1:location ?p2:location)
	(next ?f1:fuel-level ?f2:fuel-level))
(:action	drive
:parameters	(?v:vehicle ?from:location ?to:location ?fbefore:fuel-level ?fafter:fuel-level)
:precondition	(and (at ?v ?from), (accessible ?v ?from ?to), (fuel ?v ?fbefore),
	(next ?fbefore ?fafter))
:effect	(and (not (at ?v ?from)), (at ?v ?to), (not (fuel ?v ?fbefore)), (fuel ?v ?fafter))
)	
)	

delete list following the STRIPS formalism. When applied to a state, the semantics of an action specifies the condition in which the action can be applied, as well as the outcome state as a result. The pre- and postconditions of actions are expressed as logical propositions and logical connectives, where the postconditions can be split into add and delete lists. An action can be instantiated first before being applied to a state. For example, an action schema might specify load(?x ?y) which specifies the action of loading object $?x$ into vehicle $?y$. When being applied, $?x$ can be replaced by an object *Book1* whereas $?y$ can be replaced by *Truck2*, resulting in an action load(Book1 Truck2). In particular, an action a can be applied to a state s if all of the action's preconditions are members of the state s. The next state can be obtained by deleting the delete list relations and then adding the add-list relations to s. One important extension of PDDL over the STRIPS language is the ability to specify typed parameters. For example, $?x$: *vehicle* in Table 3 specifies that the variable $?x$ can *only* be bound to an object that is a vehicle.

Table 3 illustrates how PDDL can be used to model a domain in which a vehicle can move between locations [14] (where the relations preceded by "not" operator can be considered as delete-list items). There are five levels of language descriptions in PDDL 2.1, each allowing more expressivity such as consumable resource requirements. In this paper, we focus on the STRIPS level of PDDL 2.1 [14]. An example of a problem description in the vehicle domain is shown in Table 4, which includes both an initial state and a goal state.

3. 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 relations that are instantiated with actual parameters. Every action has a precondition list, which is a list of formulas that must hold in a state just before the action 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 action-learning algorithm ARMS 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 signatures consisting 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 models for some or all actions may be incomplete in that some preconditions, add and delete lists of actions may not be completely specified. In the input instances to ARMS, 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.

In each plan, the initial and goal states are replaced by two special actions whose models do not need to be learned. The first action of a plan, which is denoted by α_{init}, replaces the initial state. This action has no preconditions and has as its add list all the initial-state relations. Its delete list is empty. Similarly, the last action of a plan, which is α_{goal}, replaces the goal state. The preconditions of α_{goal} are the goal relations, but the effects are empty.

Table 4
A problem instance associated with the vehicle domain

(define	(problem vehicle-example)
(:domain	vehicle)
(:objects	truck car - vehicle
	full half empty - fuel-level
	Paris Berlin Rome Madrid - location)
(:init	(at truck Rome)
	(at car Paris)
	(fuel truck half)
	(fuel car full)
	(next full half)
	(next half empty)
	(accessible car Paris Berlin)
	(accessible car Berlin Rome)
	(accessible car Rome Madrid)
	(accessible truck Rome Paris)
	(accessible truck Rome Berlin)
	(accessible truck Berlin Paris)
)	
(:goal	(and (at truck Paris)
	(at car Rome))
)	
)	

Our objective is to learn a complete set of action definitions with precondition, add and delete lists assigned to each action schema. This collection is called an *action model*. In order to ensure the generality of the learned action models and to avoid overfitting the models to the training examples, the action models are not required to be 100% correct. Thus, we will define the notion of an error rate which is evaluated on a separate set of plans known as a test set. For each action in the test set, we replace it with the learned action model. We then define the following metrics to evaluate an action model.

We 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* with respect to an action model if according to the action model, (1) each action's preconditions hold in the state just before that action and (2) all goal propositions hold after the last action.

If a precondition of an action is not satisfied in the preceding state of an action in a plan example, then we say an error occurs, and we use $E(a) = E(a) + 1$ to count a's error in P.

Definition 2. In a plan P, one count of error is said to have occurred if an action's precondition p is not true in the state just before the action a in P. Let $E(P)$ be the total count of errors in a plan P. Then the *error rate* of P is

$$E(P) = \frac{\sum_{a \in P} E(a)}{\sum_{a \in P} |precond(a)|}$$

For a set of plans Σ_P, we can define an *error rate* of an action model with respect to the plans in Σ_P similarly as the total count of errors $E(\Sigma_P)$ in Σ_P over the total number of preconditions of all actions in Σ_P.

We can conversely define the correctness of an action model with respect to a plan or a set of plans.

Definition 3. An action model is *correct* with respect to a plan if according to the action model, the error count is zero for the plan. Similarly, an action model is said to be correct with respect to a set of plans if the error rate is zero for the set of plans.

Another aspect of the action model is that we wish to preserve the *usefulness* of all actions in the training examples. This is important because one can easily assign all goals to one of the actions in a plan while leaving the preconditions and post-conditions of all other actions empty; this model is clearly not an ideal one, even though the resulting plans are still correct. However, by doing so, most of the actions in a plan are rendered useless. In our learning framework, to ensure the learned action model is non-trivial, we first require that every action's add list is non-empty. Furthermore, we apply a basic assumption that every action's add list member in an example plan is *useful* for some later actions in the same plan. Conversely, we can define the notion of redundancy as follows.

Definition 4. An add-list element r of an action a in a plan P is *non-redundant* in P if there exists an action b ordered after a in the plan, where r is in the precondition of b, and there is no action between a and b that adds r. Otherwise, r is said to be redundant.

The intuition is that an action's add-list relation r is redundant in a plan P if this relation is not "useful" in achieving the precondition of any later actions in the plan. The number of redundant add list elements in all example plans provides a measure of how small the action model is. In the extreme case, if all relations are in the add list of all actions, then the example plans are all made trivially correct. However, in such a case, the level of redundancy for the action model is also at a maximum. Thus, it is a tricky task to balance the number of relations in an action model and the number of errors incurred by the action model on a set of training plans.

We can measure the redundancy rate of actions in a set of plan examples as follows. For every action a, let $R(a)$ be the number of add list relations that are not used to achieve a precondition of a later action in the plan. Then for a set of plans Σ_P, the redundancy rate is:

$$R(\Sigma_P) = \frac{\sum_{a \in \Sigma_P} R(a)}{\sum_{a \in \Sigma_P} |addlist(a)|} \tag{1}$$

Finally, we define our *ideal* target of learning.

Definition 5. An action model is *concise* if it is both correct and has the lowest redundancy rate among all correct action models with respect to a set of example plans Σ_P.

Intuitively, we consider an action model to be a *good* one if it explains as many plan examples as possible in the testing plan examples, and if it is the *simplest* one among all such correct models. Note that a concise model is not necessarily the smallest model among all correct action models, since it might still be possible to further reduce the number of add-list relations if one simultaneously removes some precondition relations. Such global optimization would be difficult to achieve computationally. Thus we settle for the notion of a concise model in this work.

Note also that in our problem statement, we assumed that, in our learning problem, the actions are observed while the states are not. If, however, the actions themselves are not observable, but the states are observable, then it is necessary to know whether for every pair of adjacent observed states, the same action occurred. In this case, if the state values are continuous, then the first problem is one of time-series segmentation, where the continuous signal sequence is a time series of sensor values. The segmentation problem is one in which one determines the beginning and end of each action. Subsequently, one can still apply our learning algorithm to learn the actions' preconditions and effects. An initial attempt at solving this problem is [47,48]. If the states are discrete-valued, then the problem is one of predicting a minimal set of actions and the associated models, such that the number of actions, sizes of precondition and effect sets, as well as the model's error rate, are minimized according to the minimal description length principle [20].

4. The ARMS algorithm

4.1. Overview

Given a set of correct example plans Σ_P, we wish to uncover the constraints and use them to confine the space of learned models. To illustrate our ideas, consider a simple example. Let there be a single plan P in which the initial state contains two relations $p1$, $p2$, and the goal has one relation g. Let P contain two actions $a1$ and $a2$ where $a1$ is ordered before $a2$. There is an additional relation $p3$ that is not present in the initial state.

To learn action models for $a1$ and $a2$, we can consider the following possibilities:

(1) $p1$ and $p2$ both belong to the precondition list of $a1$, and $a2$ adds the goal g. $a1$'s add and delete lists and $a2$'s precondition list are all empty.
The problem with this action model is that $a1$ becomes a redundant action, and the order between $a1$ and $a2$ remains unexplained.
(2) An alternative model is to assign $p1$ to the precondition of $a1$ and $p2$ to the precondition of $a2$. $a1$'s add list contains $p3$ which is a precondition of $a2$. Finally, $a2$ adds the goal g. In this model, both $a1$ and $a2$'s delete lists are empty.
In the second model, both actions are non-redundant. In addition, the explained plan is correct. Therefore, this is a preferred model.

In general, for a given set of plans Σ_P, there may be a large number of potential action models. To find the models that have low error and redundancy rates, we uncover a number of constraints from the plans. The first type of constraints constrain the preconditions, add and delete lists for any single actions. These constraints must be universally true. Therefore, the precondition of an action cannot appear in the add list of an action, and the delete list must be a subset of the precondition list. This constraint is known as the action constraint. The second type of constraint restricts the reason why an action $a1$ occurs before another action $a2$ in a plan P. This may be because $a1$ achieves a precondition p for $a2$, or $a1$ deletes a relation r that is added back by $a2$, or both $a1$ and $a2$ require a common precondition r. This type of constraint applies to two or more actions in a plan, and is called *plan constraints*. Finally, if we observe a relation r to be true between two actions $a1$ and $a2$, we can give r higher priority in assigning it to the add list of $a1$ and precondition list of $a2$. We also wish to limit the size of preconditions, add and delete lists of all actions to be no more than a predefined number k. We call these types of preference constraints *information constraints*.

Because for any example plan set $Sigma_P$, there may be a large number of constraints that can be uncovered, we prefer to use the statistical information to generate a relatively small set of constraints as the basis of learning. In particular, we apply frequent set mining to find the frequent subsets of actions in which to apply the plan and information constraints. We describe the algorithms in detail in the next section.

4.2. Algorithm and constraints

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:

The ARMS Algorithm
Input: Variables, Constants and Relations in the Domain; A set of correct plan examples.
Output: A set of action models Θ
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 models in Θ by the set of empty action schemata.
Step 2 For the unexplained actions in Λ, build a set of information and action constraints based on individual actions and call it Ω. Apply a frequent-set mining algorithm [1] to find the frequent sets of *connected* actions and relations (see the next subsection). Here *connected* means the actions and relations must share some common parameters. Let the frequent set of action-relation pairs be Σ.
Step 3 Build a weighted maximum satisfiability representation Γ based on Ω and Σ based on the constraints and frequency information uncovered in Step 2.
Step 4 Solve Γ using a weighted MAX-SAT solver. In the experiments, we used both http://www.nmt.edu/~borchers/maxsat.html and the MaxWalkSat solver [26,39]. Section 6.3.4 provides more details on the application of MaxWalkSat.
Step 5 Select a set A of learned action models in Λ with the highest weights. Update Λ by $\Lambda - A$. Update Θ by adding A. If Λ is not empty, go to Step 2.
Step 7 Output the action models Θ.

The algorithm starts by initializing the plans by replacing the actual parameters of the actions by variables of the same types. This ensures that we learn action models for the schemata rather than for the individual instantiated actions. Subsequently, the algorithm iteratively builds a weighted MAX-SAT representation and solves it. In each iteration, a few more actions are explained and are removed from the incomplete action set Λ. The learned action models in the middle of the program help reduce the number of clauses in the SAT problem. ARMS terminates when all action schemata in the example plans are learned.

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

4.3. Step 1: Initialize plans and variables

A plan example consists of a sequence of action instances. We *convert* all such plans by substituting all occurrences of an instantiated object in every action instance with the variables of the same type. If the object has multiple types, we generate a clause to represent each possible type for the object. For example, if an object o has two types *Block* and *Table*, the clause becomes: $\{(?o = \textit{Block})$ or $(?o = \textit{Table})\}$. We then extract from the example plans all sets of actions that are *connected* to each other; two actions $a1$ and $a2$ are said to be *connected* if their parameter-type list has non-empty intersection. The parameter mapping $\{?x1 = ?x2, \ldots\}$ is called a connector.

4.4. Step 2: Build action and plan constraints

As introduced in Section 2, a weighted MAX-SAT problem consists of a set of clauses representing their conjunction, where each clause is associated with a weight value representing the priority in satisfying the constraint. Given a weighted MAX-SAT problem, a weighted MAX-SAT solver finds a solution by maximizing the sum of the weight values associated with the satisfied clauses.

In the ARMS system, we have four kinds of constraints to satisfy, representing three types of clauses. They are action, information and plan and relation constraints.

4.4.1. Action constraints

Action constraints are imposed on individual actions. These constraints are derived from the general axioms of correct action representations. A relation r is said to be *relevant* to an action a if they are the same parameter type. Let pre_i, add_i and del_i represent a_i's precondition list, add-list and delete list. The general action constraints are translated into the following clauses for building the SAT:

(1) (Constraint A.1) The intersection of the precondition and add lists of all actions must be empty.

$$pre_i \cap add_i = \phi.$$

(2) (Constraint A.2) In addition, if an action's delete list includes a relation, this relation is in the action's precondition list. Thus, for every action, we require that the delete list is a subset of the precondition list.

$$del_i \subseteq pre_i.$$

To illustrate action constraints, consider the following example.

Example 1. Consider the action signature load(?x:hoist ?y:crate ?z:truck ?p:place) in the Depot domain of International Planning Competition (IPC, see Table 2). From the domain description, the possible relations of the action signature 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 the last action consists of (on ?y:crate ?s:surface). Suppose that the primary effect of the action load(?x ?y ?z ?p) is (in ?y ?z). For this action, the action constraints are given as follows:

- The preconditions include all possible relations that are joined together by a disjunction. The possible relations of the add and delete list are (lifting ?x ?y), (at ?y ?p), (in ?y ?z), (clear ?y), or (at ?z ?p).
- Constraint A.1 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.2 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)$\in pre_i$

4.4.2. *Information constraints*

The information constraints are used to explain why the optionally observed intermediate states exist in a plan. The constraints thus derived are given high priority because they need not be guessed.

Suppose we observe a relation p to be true between two actions a_n and a_{n+1}, and p, $a_{i_1}, \ldots,$ and a_{i_k} share the same parameter types. We can represent this fact by the following clauses, given that $a_{i_1}, \ldots,$ and a_{i_k} appear in that order.

- (Constraint I.1) The relation p must be generated by an action $a_{i_k}(0 \leqslant i_k \leqslant n)$, that is, p is selected to be in the add-list of a_{i_k}. $p \in (add_{i_1} \cup add_{i_2} \cup \cdots \cup add_{i_k})$, where $\cup$ means logical "or".
- (Constraint I.2) The last action a_{ik} must not delete the relation p; that is, p must not be selected to be in the delete list of a_{ik}: $p \notin del_{ik}$.

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

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

Finally, we consider a 'soft' type of information constraint to define pairs of relations and actions where the relation r frequently occurs before action a. This is often the case when the initial states of a plan contain many relations, some of which often occurs before the first action of a plan. In this case, we predict that these facts are preconditions of the action. This constraint applies also to other parts of a plan, where the partially observable states are in the middle of a plan.

- (Constraint I.3) We define the weight value of a relation-action pair (p, a) as the occurrence probability of this pair in all plan examples. If the probability of a relation-action pair is higher than the probability threshold θ, then we set a corresponding relation constraint $p \in \mathtt{PRECOND}_a$, which receives a weight value equal to its prior probability.

Example 3. Consider the example in Table 2. A relation-action pair (clear c0), lift(h1 c0 p1 ds0) (sharing a parameter $\{c0\}$) from Plan1 and relation-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 occurrence count for (clear ?y), lift(?x ?y ?z ?p) with the parameter label $?y$ is at least two. Later, in Section 4.4.3, we will define this count to be the *support count* for frequent-sequence analysis. Table 5 shows examples of the I.3 type of constraints.

Table 5
Examples soft information constraints

Parameters	Information constraints
{?y}	(clear ?y) $\in pre_{lift}$
{?x, ?p}	(at ?x ?p) $\in pre_{lift}$
{?x }	(available ?x) $\in pre_{lift}$
{?z, ?p}	(at ?z ?p) $\in pre_{lift}$
{?y, ?p}	(at ?y ?p) $\in pre_{lift}$
{?y, ?z}	(on ?y ?z) $\in pre_{lift}$
{?x, ?y}	(at ?x ?y) $\in pre_{drive}$
{?y, ?z}	(on ?y ?z) $\in add_{drop}$

4.4.3. Plan constraints

The plan constraints represent the relationship between actions in a plan to ensure that a plan is correct and that the actions' add list relations are not redundant. First, for each action in a plan, we generate a constraint for the requirement for a plan to be correct. The first of the plan constraints is as follows:

- (Constraint P.1) Every precondition p of every action b must be in the add list of a preceding action a and is not deleted by any actions between a and b.
- (Constraint P.2) In addition, at least one relation r in the add list of an action must be useful in achieving a precondition of a later action. That is, for every action a, an add list relation r must be in the precondition of a later action b, and there is no other action between a and b that either adds or deletes r.

While constraints P1 and P2 provide the general guiding principle for ensuring a plan's correctness, in practice there are too many instantiations of these constraints. This is because for a given action's precondition, any preceding action could serve as an "establisher" of the relation. To ensure efficiency, we design heuristics that consider the reasons that can explain why two actions frequently co-exist. These heuristics allow us to restrict ourselves to only a small subset of frequent action pairs. When explaining which action adds a relation for the precondition of which other actions, we give higher priority to these actions.

Our heuristic is based on the notion of association-rule mining in data mining area [1]. The following constraints P.3–P.6 are applied to a set of action pairs that co-occur frequently. The rationale of these constraints is that if a pair of actions occur frequently enough in the plan examples, there must be a reason for their frequent co-existence. Thus, the possible reasons for this co-occurrence can be enumerated to uncover these reasons.

Our first task is to find frequent action pairs that occur in the plans. We apply the Apriori algorithm [1] to find frequent action pairs $\langle a_i, a_j\rangle$ where $0 \leqslant i < j \leqslant n-1$ are the indexes of the actions. To capture how often a frequent pair occurs in the plan examples in a training set of plan examples, we make the following definitions.

Definition 6. Let S be an action pair $\langle a_i, a_j\rangle$, $0 \leqslant i < j \leqslant n-1$. The *support count* of the pair S in a set of plans Σ_P is the number of times that this pair occurs in the plans from Σ_P.

The support count is an integer value. We can equivalently talk about the support rate which is the percentage of the occurring pairs:

Definition 7. Let S be an action pair $\langle a_i, a_j\rangle$, $0 \leqslant i < j \leqslant n-1$, where n is the number of actions in a plan. The *support rate* of the pair S in a set of plans Σ_P is the percentage that this pair occurs in Σ_P

$$\mathit{support\ rate}\ \big(\langle a_i, a_j\rangle\big) = \frac{\mathit{support\ count}\ (\langle a_i, a_j\rangle)}{Z_{\Sigma_P}}$$

where Z_{Σ_P} is the total number of action pairs in Σ_P.

'Frequent pairs' is a relative term. Here we follow the data mining literature to define a probability threshold θ, so that all action pairs whose support rate is above or equal to θ are considered frequent enough. We apply the subsequent constraints only to explain the frequent action pairs. Of course, θ has to be defined differently for each domain, and this is indeed a free parameter we wish to tune in our later experimental part. Note that unlike general association rules, 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. These pairs are to be explained by the learning system later.

Let there be an action pair $\langle a_i, a_j\rangle, 0 \leqslant i < j \leqslant n-1$.

- (Plan constraint P.3) One of the relevant relations p must be chosen to be in the preconditions of both a_i and a_j, but not in the delete list of a_i,

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

- (Constraint P.4) The first action a_i adds a relevant relation that is in the precondition list of the second action a_j in the pair,

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

- (Constraint P.5) A relevant relation p that is deleted by the first action a_i is added by a_j. The second clause is designed for the event when an action re-establishes a fact that is deleted by a previous action.

$$\exists p\ (p \in (del_i \cap add_j))$$

- Let pre_i, add_i and del_i represent a_i's precondition list, add-list and del-list, respectively. The above plan constraints can be combined into one constraint $\Phi(a_i, a_j)$ in ARMS. $\Phi(a_i, a_j)$ is restated as:

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

As an example, consider the Depot domain (Table 2).

Example 4. Suppose that lift(?x1:hoist ?y1:crate ?z1:surface ?p1:place), load(?x2:hoist ?y2:crate ?z2:truck ?p2:place)) is a frequent pair. We explain the connection between the two actions, lift and load, by selecting a candidate relation. The relevant parameters that share the same type are ?x1 = ?x2, ?y1 = ?y2, ?p1 = ?p2. The relations that refer to these parameters are (at ?x ?p), (available ?x), (lifting ?x ?y), (at ?y ?p), and (clear ?y). From this action pair, the SAT clauses can be constructed as follows (we use ?x to represent the parameter type "hoist", ?y to represent "crate", and ?p to represent "place"):

- At least one relation among (at ?x ?p), (available ?x), (lifting ?x ?y), (at ?y ?p), and (clear ?y) is selected to explain the connection between lift(?x ?y ?z ?p) and load(?x ?y ?z ?p).
- If $f(?x)$, where $f(?x)$ can be (at ?x ?p), (available ?x), (lifting ?x ?y), (at ?y ?p), and (clear ?y), is selected to explain the connection between lift(?x ?y ?z ?p) and load(?x ?y ?z ?p), then one of the following is true: (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).

4.5. Step 3: Build and solve a weighted MAX-SAT problem

In solving a weighted MAX-SAT 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 MAX-SAT problem, we apply the following heuristics:

Action constraints Every action constraint receives a constant weight $W_A(a)$ for an action a. The weight for action constraints is set to be higher than the weight of information constraints.

Information constraints Every partial information constraint receives a constant weight $W_I(r)$ for a relation r. The constant assignment is empirically determined too, but they are generally the highest in all constraints' weights.

Table 6
Examples of highly supported information constraints

Label	Information constraints	Support count	Support rate (%)
{?y}	(clear ?y) $\in pre_{lift}$	3	100
{?x, ?p}	(at ?x ?p) $\in pre_{lift}$	3	100
{?x}	(available ?x) $\in pre_{lift}$	3	100
{?z, ?p}	(at ?z ?p) $\in pre_{lift}$	3	100
{?y, ?p}	(at ?y ?p) $\in pre_{lift}$	3	100
{?y, ?z}	(on ?y ?z) $\in pre_{lift}$	3	100
{?x, ?y}	(at ?x ?y) $\in pre_{drive}$	1	33
{?y, ?z}	(on ?y ?z) $\in add_{drop}$	3	100

For information constraints $I.3$, we define the weight value of a relation-action pair (p, a) as the occurrence probability of this pair in all plan examples. If the probability of a relation-action pair is higher than the probability threshold θ, then we set a corresponding relation constraint $p \in \mathtt{PRECOND}_a$, which receives a weight value equal to its prior probability. If not, the corresponding relation constraint receives a constant weight of a lower value which is determined empirically. This assignment corresponds to a new type of heuristic constraints, which is especially useful for partial information constraints.

Example 5. Consider the example in Table 2. A relation-action pair (clear c0), lift(h1 c0 p1 ds0) (sharing a parameter $\{c0\}$) from Plan1 and relation-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 count for (clear ?y), lift(?x ?y ?z ?p) with the parameter label $?y$ is at least two. Table 6 shows all information constraints along with their support rate values in the previous example.

Plan constraints The weight value of a plan constraint $W_P(a1, a2)$ is higher than the probability threshold (or a minimal support-rate value) θ. This value is set equal to the prior probability of the action pair $(a1, a2)$.

In the experiments, we will vary the value of the threshold θ to verify the effectiveness of algorithm. When considering subsequences of actions from example plans, we only consider those sequences whose supports are over θ.

Example 6. Consider the example in Table 2. An action sequence ⟨lift(h1 c0 p1 ds0), load(h1 c0 t0 ds0)⟩ that shares parameters {h1, c0, ds0} from Plan1 and an action sequence ⟨lift(h1 c1 c0 ds0), load(h1 c1 t0 ds0)⟩ that shares parameters {h1, c1, 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} indicates that the parameters ?x1, ?y1 and ?p1 in action lift(?x1 ?y1 ?z1 ?p1) are the same as the parameters ?x2, ?y2 and ?p2 in action load(?x2

Table 7
Examples of all action constraints

Label	Plan constraints	Support count	Support rate (%)
{?x1 = ?x2, ?y1 = ?y2, ?p1 = ?p2}	Φ(lift, load)	5	100
{?z1 = ?x2, ?p1 = ?y2}	Φ(load, drive)	4	80
{?x1 = ?z2, ?z1 = ?p2}	Φ(drive, unload)	4	80
{?y1 = ?y2, ?p1 = ?p2}	Φ(unload, drop)	5	100
{?x1 = ?x2, ?p1 = ?p2}	Φ(load, lift)	2	40
{?x1 = ?x2, ?p1 = ?p2}	Φ(drop, unload)	1	20
{?x1 = ?z2, ?z1 = ?p2}	Φ(drive, load)	1	20
{?y1 = ?p2}	Φ(drive, load)	1	20
{?p1 = ?p2 = ?y3 }	Φ(lift, load, drive)	4	80
{?z1 = ?x2 = ?z3 }	Φ(load, drive, unload)	4	80
{?z1 = ?p2 = ?p3 }	Φ(drive, unload, drop),	4	80
{?p1 = ?p2 = ?p3 = ?y4}	Φ(load, lift, load, drive)	2	40
{?z1 = ?p2 = ?p3 = ?p4 }	Φ(drive, unload, drop, unload)	1	20

?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 7 shows all plan constraints along with their support count and rate values.

4.6. Step 5: Update initial states and plans

ARMS starts with a set Λ of incomplete action models. As more action schemata are fully learned, that is, when the maximum size of the action's preconditions and add/delete lists reach an upper bound value U, the action is considered as learned. All learned actions are removed from the set Λ in this step. If a learned action appears in the beginning of a plan, ARMS updates the initial state by executing the actions in the current initial state, which produces a new initial state.

5. Properties of the ARMS algorithm

5.1. Formal properties and evaluation metrics

Given a set of plan examples, we can find a large number of action models. Some models are superior to others. An advantage of ARMS is that when the action and plan constraints are satisfied by a set of plans, the plan is correct.

Theorem 1. *For a given set of training example plans Σ_P, when the action constraints* (A.1) *and* (A.2)*, and the plan constraint* (P.1) *are satisfied for all actions, the plans in P are correct.*

Proof. This theorem follows from the definition of plan correctness, which states that a plan is correct if all preconditions of all actions are true in the state just before that action (the goal included). Consider any single plan P in the plan set Σ_P. For an action b, when the plan constraint (P.1) is satisfied, each of b's preconditions $r \in \mathtt{PRECOND}(b)$ is in the add list of some action a before b, and no other actions between a and b delete r. The action constraints (A.1) and (A.2) defined in Section 4.4.1 ensure that r is not deleted by a itself. These conditions therefore ensure that r is true just before b. Thus the plan is correct. $\square$

The following theorem establishes that in the ideal case, no relation in the add list of an action model is redundant in any of the example plans.

Theorem 2. *For a given set of training example plans Σ_P, when the plan constraint* (P.2) *is satisfied by all actions, the learned action models have zero redundancy rate with respect to Σ_P (thus, they are* concise*).*

Proof. We wish show that in each example plan, every add list relation is used to establish some precondition of a later action in the same plan. This is ensured by the constraint (P.2), which states that every add list member of an action a is useful in adding a relation r that is a member of a precondition of a later action b, and that no other action between a and b adds or deletes r. Thus, the redundancy rate is zero for the action model on the training plans Σ_P. $\square$

It is important to guarantee that a learned action model is correct and has a low redundancy rate. In this case, we consider the learned model to be approximately *concise* (see Definition 5. To achieve this objective, ARMS uses a greedy algorithm by explaining only the action pairs that are sufficiently frequent, where the concept of frequency is defined by a probability lower bound value θ). As a result, it is still possible that some preconditions of some actions in the learned model are not explained, if the preceding actions do not appear frequently enough. We will use the error rates $E(P)$ to estimate the degree of correctness of a plan, and the redundancy rate $R(P)$ to measure the degree of redundancy in the training plans. We will show these two metrics as a function of several training plan parameters in the next section.

Finally, the number of clauses generated by ARMS is polynomial in the number of relations in the domain, plan sizes, the number of training plans and the number of actions.

Theorem 3. *The number of clauses generated by* ARMS *for a set of example plans is polynomial in the number of relations, action schemata and the size of the example plans used for training.*

Proof. Because the process of generating the constraints follows a finite sequence of steps, the complexity of ARMS mainly depends on the number of clauses and variables in the SAT problem.

Consider a planning domain in which there are A actions in each plan and N plans. Let U be upper bound on the number of the relations in the preconditions and postconditions of each action. Then the number of action constraints is $\mathrm{O}(A * U * N)$. The number of plan constraints is $\mathrm{O}(A^2 * U * N)$. The number of information constraints is $\mathrm{O}(U * A * N)$. Thus, the total number of clauses is bounded by $\mathrm{O}(A^2 * N * A * U)$. □

Note that due to the local-search nature of the ARMS algorithm, we cannot guarantee that the learned action model is the smallest one theoretically. However, since the weighted MAX-SAT algorithm generates a solution for a SAT problem parsimoniously, the action model generated is generally small, thus the learned model is approximately *simple*. In the next section, we empirically verify the redundancy rates of the learned models.

5.2. Relation to Markov logic networks

In the related works section, we have briefly mentioned that there exists a strong relation between action-model learning and Markov logic networks (MLN). MLN combines Markov networks and first order logic in a unified learning and inference framework [9,35,36]. In a Markov network, there is a set of variables that corresponds to the network nodes, $X = (X_i, \dots, i = 1 \dots n)$, where the joint probability distribution is defined in a log-linear model. Over this network, a first order logic framework is represented by including the logic constructs that consist of constants, predicates and truth value assignments. In addition, a knowledge base is given that includes a set of first order logic formulae.

In the formal definition of a MLN, there is a set of binary valued nodes for each possible grounding of each predicate appearing in the MLN [9,35,36]. There is also a set of features corresponding to each formula F_i in the network, where the value of each F_i is binary. Associated with a formula is a weight value. The higher the weight, the stronger the features are. A strong feature of the MLN formalism is its ability to perform learning, where there are two tasks to be performed [36]. The first task is to learn the structure of the Markov network, which can be solved using an inductive logic programming approach (ILP). The second task is to learn the weight parameters of a MLN, which can be learned using methods such as conjugate gradient or iterative scaling. Once learned, the MLN can be used to accomplish a number of interesting inference tasks. For example, in collective classification, MLN can be applied to predict the classes of related objects such as linked Web pages, by considering the objects together.

The ARMS algorithm can be considered as a special case of the MLN algorithm. In particular, we invent three predicates as follows:

- We invent a new predicate InPrecond such that, for a literal P and action A, InPrecond takes P and A as arguments in the form of InPrecond(P, A) to denote the fact that the literal P is assigned to the precondition of the action A.
- We invent a new predicate InAdd such that, for a literal E and action A, InAdd takes E and A as arguments in the form of InAdd(E, A) to denote the fact that the literal E is assigned to the effects of the action A.
- Similarly, we define InDelete(E, A) for the delete list items.

Then, we can convert all the constraints mentioned in Section 4.2 by these new predicates. For example, the action constraint (A.1) (Section 4.4.1) can be represented as a knowledge-base (KB) formula:

$$\forall P \in \mathit{Literals}, A \in \mathit{Actions}.\mathtt{InPrecond}(P, A) \Rightarrow \neg\mathtt{InAdd}(P, A)$$

which states that the intersection of preconditions and add list of all actions are empty.

We can represent the action-model learning problem as a MLN learning problem, as follows. As stated above, we invent new predicates InPrecond, InAdd and InDelete that apply to two types of objects: relations and actions. Relations are constructed using the domain specific predicates, variables and constants such as (on ?x,?y). Relations between predicates include logical axioms that encode specific constraints in a problem domain. For example, in the blocks world, an axiom states that the predicate (clear ?x) = True preclude that there exists an object ?y, such that (on ?y,?x) = True. These axioms, as well as the constraints, form the relations between the nodes in the MLN. The weights of the *hard* constraints, such as the above example, are set to be the highest. The action, information and plan constraints receive their weights according to the procedure in Section 4.4. Then, we can use an algorithm for solving the corresponding weighted MAX-SAT problem for obtaining an approximate solution, in which the sum of all weights are maximized within limits imposed by the computational resources. In the experimental section, we describe two implementations of ARMS using the systems MaxSatSolver and MaxWalkSat, respectively. We will report that the performance of these two weighted MAX-SAT solvers are similar in terms of the approximate solutions found and the CPU time spent.

6. Experimental results

In this section we assess the effectiveness of the learned action model empirically. An action model may be incomplete and error-prone with respect to a set of example plans. A previous evaluation method [16] applied a planning system to the incomplete action model, and then assessed the model's degrees of correctness. In this work, we have an available set of example plans that can be split into training and testing sets. We adapt the cross-validation evaluation method by making full use of all available example plans. We split the given 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 is defined for assessing the model. We take each test plan in the test data set in turn and evaluate whether the test plans are correct and non-redundant when modelled by the learned action model.

6.1. Experimental setup and planning domains

In order to evaluate ARMS, we generated example plans using an existing planning tool from the planning domains in International Planning Competition 2002.[1] The example plans are generated using the *MIPS* planner.[2] In each domain, we first generated 200 plan examples. We then applied a five-fold cross-validation by dividing the examples into five parts of equal size. We selected four of the five parts which consist of *160* plan examples as the training set from four folds and use the remaining fold with *40* separate plan examples as the test set. This is repeated five times using a different fold for the test data. We plot the mean values of various metrics with the associated error bars (corresponding to 95% confidence intervals) obtained from the five-fold cross-validation experiments. We ran all of our experiments on a personal computer with 768 MB of memory and a Pentium Mobile Processor 1.7 GHz CPU, with the Linux operating system. Also, in all experiments, we set the upper bound U for all actions according to the maximum such value in the actual domain description. This parameter can be experimentally found by varying U from a small value to a large value, and checking the resultant error and redundancy rates of the action models in all test plans. The best such value can be determined this way empirically.

The six domains are the Depots, Driverlog, Zenotravel, Satellite, Rover and Freecell planning domains. Features of the six domains are summarized in Table 8. In this table, we summarize the number of action schemata in each domain (# actions), the number of different predicates in each domain (# relations), the maximum number of relations in a precondition list or a postcondition list of each domain (Max Pre/Eff), the maximum number of relations in the

Table 8
Features of the problem domains

Domain features	Domain names					
	Depots	Driverlog	Zenotravel	Satellite	Rover	Freecell
# actions	5	6	5	5	5	10
# relations	6	5	5	8	25	11
Max Pre/Eff	4	3	4	6	6	10
Max Initial	24	48	19	28	63	149
Max Goals	3	8	8	5	6	5
# Plans	200	200	200	200	200	200
Avg. Length	10	26	24	16	23	27

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

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

Table 9
The learned action model for the Depot 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)**

Table 10
The learned action model for the Driverlog domain, with $\theta = 10\%$

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	board-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 ?loc-to)
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)

initial state (Max Initial), and goal state (Max Goals) in the training examples, the number of example plans used for training and testing, and finally, the average number of actions in the training plans (Avg. Length). This summary table serves to provide a sense of the complexity of the learning task in each domain. As we can see, the Freecell domain has the most demanding learning task, with 10 action schemata, 11 relations and long plan lengths. The Depot domain

Table 11
Varying the probability threshold in the planning domains (Depots)

Probability threshold (%)	Depots			
	E	R	CPU (sec)	Clauses
0	0.16	0.18	9.6 ± 0.5	6607 ± 4
1	0.19	0.20	9 ± 0.7	4292 ± 3
2	0.12 ± 0.03	0.17	7.8 ± 0.4	2967 ± 725
3	0.24 ± 0.06	0.24 ± 004	7	705 ± 884
4	0.27	0.25	7	310 ± 3
5	0.27	0.25	6.8 ± 0.4	310 ± 3
6	0.26 ± 0.04	0.22 ± 006	7	284 ± 57
7	0.19	0.11	7.2 ± 1.1	180 ± 3
8	0.19	0.11	7.2 ± 0.4	180 ± 3
9	0.19	0.11	7.2 ± 0.4	180 ± 3
10	0.19	0.11	7.4 ± 0.5	180 ± 3
30	0.19	0.11	7.4 ± 0.5	180 ± 3
50	0.19	0.11	7.4 ± 0.5	180 ± 3
70	0.19	0.11	7.4 ± 0.5	180 ± 3

Table 12
Varying the probability threshold in the planning domains (Driverlog)

Probability threshold (%)	Driverlog			
	E	R	CPU (sec)	Clauses
0	0.09	0.14 ± 0.01	53.4 ± 5.5	349 ± 3
1	0.08	0.12 ± 001	51.4 ± 3.0	312 ± 4
2	0.06	0.14	42.8 ± 2.5	138 ± 23
3	0.05	0.06 ± 0.05	41.4 ± 2.3	70 ± 19
4	0.05	0.04	34.8 ± 14.6	61
5	0.05	0.04	27 ± 16.5	61
6	0.05	0.04	20.6 ± 16.3	61
7	0.05	0.04	21.2 ± 17.2	59 ± 4
8	0.05	0.04	21.4 ± 17.0	41 ± 17
9	0.05	0.04	21.4 ± 17.0	41 ± 17
10	0.05	0.04	21.4 ± 17.4	41 ± 17
30	0.05	0.04	21.4 ± 17.4	41 ± 17
50	0.05	0.04	21.4 ± 17.4	41 ± 17
70	0.05	0.04	21.4 ± 17.4	41 ± 17

Table 13
Varying the probability threshold in the planning domains (Zenotravel)

Probability threshold (%)	Zenotravel			
	E	R	CPU (sec)	Clauses
0	0	0.13	7 ± 0.7	103 ± 2
1	0	0.13	6.8 ± 0.4	69 ± 2
2	0	0.13	7.2 ± 0.4	62 ± 6
3	0	0.13	7.2 ± 0.8	52 ± 6
4	0	0.13	7.2 ± 0.4	40
5	0	0.13	7.6 ± 0.5	40
6	0	0.13	6.8 ± 0.4	40
7	0	0.13	7.2 ± 0.4	40
8	0	0.10 ± 0.04	7.8 ± 0.4	40
9	0	0.09 ± 0.04	7 ± 0.7	40
10	0	0.09 ± 0.04	7	40
30	0	0.09 ± 0.04	7	40
50	0	0.09 ± 0.04	7	40
70	0	0.09 ± 0.04	7	40

is one of the simplest in contrast. For all domains, we generated 200 example plans, which are subsequently split into training and testing sets.

In our experiments, the following independent variables are varied:

Table 14
Varying the probability threshold in the planning domains (Satellite)

Probability threshold (%)	Satellite			
	E	R	CPU (sec)	Clauses
0	0.21 ± 0.08	0.07 ± 0.05	6	688 ± 8
1	0.23 ± 0.04	0.05 ± 0.02	5.8 ± 0.4	140 ± 8
2	0.26 ± 0.01	0.03	5.8 ± 0.4	134
3	0.26 ± 0.01	0.03	5.6 ± 0.5	134
4	0.25 ± 0.03	0.04 ± 0.02	5.4 ± 0.5	137 ± 7
5	0.21 ± 0.03	0.07 ± 0.02	5.4 ± 0.5	92 ± 23
6	0.20	0.07	5.4 ± 0.5	82
7	0.20	0.07	5.8 ± 0.4	82
8	0.20	0.07	5.2 ± 0.4	82
9	0.23 ± 0.07	0.07	4.8 ± 1.6	85 ± 6
10	0.26 ± 0.09	0.07 ± 0.01	4 ± 1.9	87 ± 7
50	0.26 ± 0.09	0.07 ± 0.01	4 ± 1.9	87 ± 7
70	0.26 ± 0.09	0.07 ± 0.01	4 ± 1.9	87 ± 7

Table 15
Varying the probability threshold in the planning domains (Rover)

Probability threshold (%)	Rover			
	E	R	CPU (sec)	Clauses
60	0.68	0.07 ± 0.02	224 ± 17	62700 ± 6774
61	0.66 ± 0.04	0.06 ± 0.01	217 ± 11	51078 ± 13355
62	0.67 ± 0.02	0.06 ± 0.02	210 ± 21	51078 ± 13355
63	0.67 ± 0.02	0.07 ± 0.02	192 ± 15	43368 ± 13355
64	0.66 ± 0.01	0.08	181 ± 1	35657
65	0.66 ± 0.01	0.08	176 ± 12	29734 ± 10259
66	0.66 ± 0.01	0.08	167 ± 11	23811 ± 10259
67	0.67	0.09 ± 0.01	166 ± 9	26762 ± 15371
68	0.67 ± 0.01	0.08	172 ± 8	23811 ± 10259
69	0.67 ± 0.01	0.08	169 ± 2	17888
70	0.67 ± 0.01	0.08	168 ± 4	17887 ± 1
75	0.67 ± 0.01	0.08	168 ± 4	17887 ± 1
80	0.67 ± 0.01	0.08	168 ± 4	17887 ± 1

Table 16
Varying the probability threshold in the planning domains (Freecell)

Probability threshold (%)	Freecell			
	E	R	CPU (sec)	Clauses
60	0.47 ± 0.01	0.47 ± 0.01	388 ± 6	91 ± 1
61	0.47 ± 0.01	0.47 ± 0.01	387 ± 6	91 ± 1
62	0.47 ± 0.01	0.47 ± 0.01	390 ± 5	91 ± 1
63	0.47 ± 0.01	0.47 ± 0.01	390 ± 5	91 ± 1
64	0.47 ± 0.01	0.47 ± 0.01	390 ± 6	91 ± 1
65	0.47 ± 0.01	0.47 ± 0.01	390 ± 5	91 ± 1
66	0.47 ± 0.01	0.47 ± 0.01	390 ± 5	91 ± 1
67	0.47 ± 0.01	0.47 ± 0.01	390 ± 6	91 ± 1
68	0.47 ± 0.01	0.47 ± 0.01	390 ± 5	91 ± 1
69	0.47 ± 0.01	0.47 ± 0.01	390 ± 5	91 ± 1
70	0.47 ± 0.01	0.47 ± 0.01	389 ± 5	91 ± 1
75	0.47 ± 0.01	0.47 ± 0.01	389 ± 5	91 ± 1
80	0.47 ± 0.01	0.47 ± 0.01	389 ± 5	91 ± 1

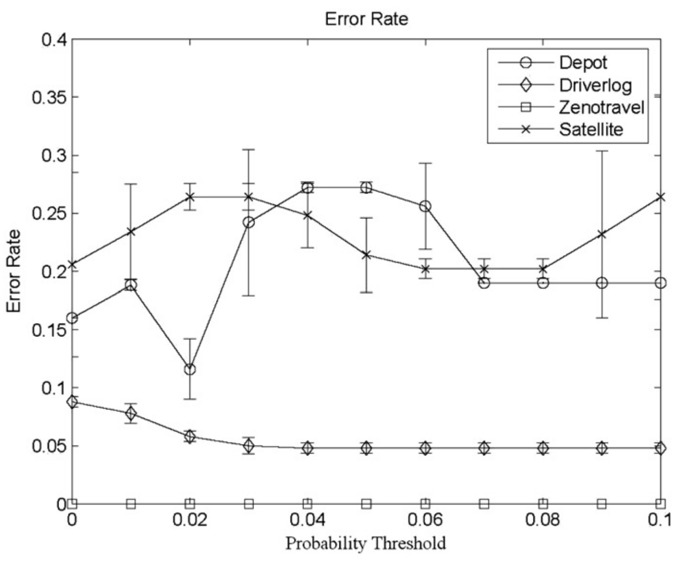

Fig. 1. Varying the probability threshold in four planning domains (Error Rate).

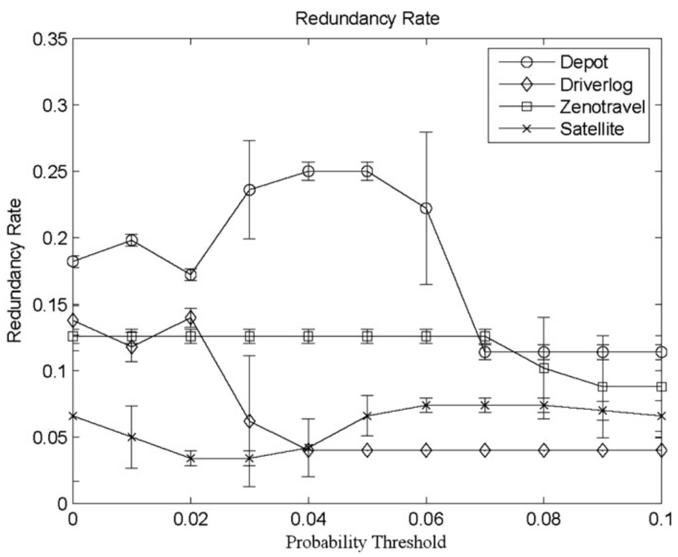

Fig. 2. Varying the probability threshold in four planning domains (Redundancy Rate).

The probability threshold θ This is used to control the frequent action pairs used in enforcing the plan constraints. When θ increases from zero to 100%, we expect to have fewer clauses in our weighted MAX-SAT formulation.

The degree of partial information This is the amount of relational information that we can observe between the actions in the training plans (partial information represented in percentage of the original states). This variable is defined to test how the ARMS algorithm might benefit from some partial information that can be occasionally observed in the middle of plans. Let p be a value that ranges from zero to 100%. We use p to control the amount of partial information that can be observed after an action, in terms of a total of $p * N$ relations where N is the total number of relations in a plan's actions after summing up their preconditions, add and delete lists together. When p ranges from zero to 100%, the amount of available state information increases.

The plan length This is the average number of actions in the example plans. These three variables are varied from small to large to test the performance of ARMS.

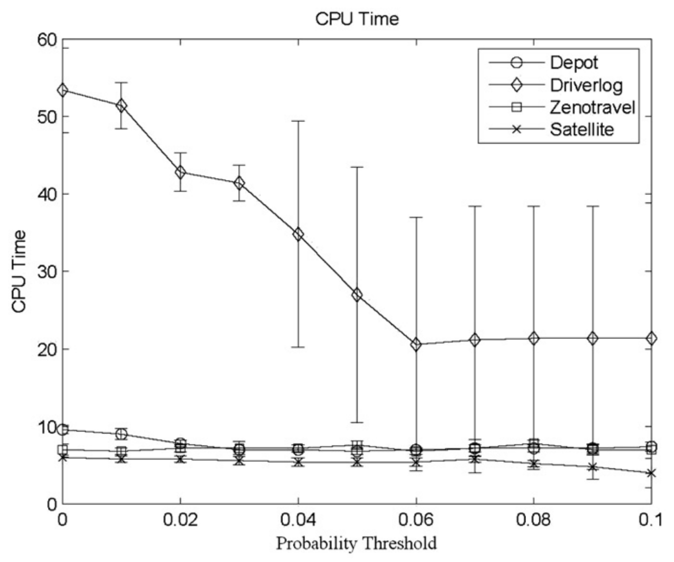

Fig. 3. Varying the probability threshold in four planning domains (CPU Time).

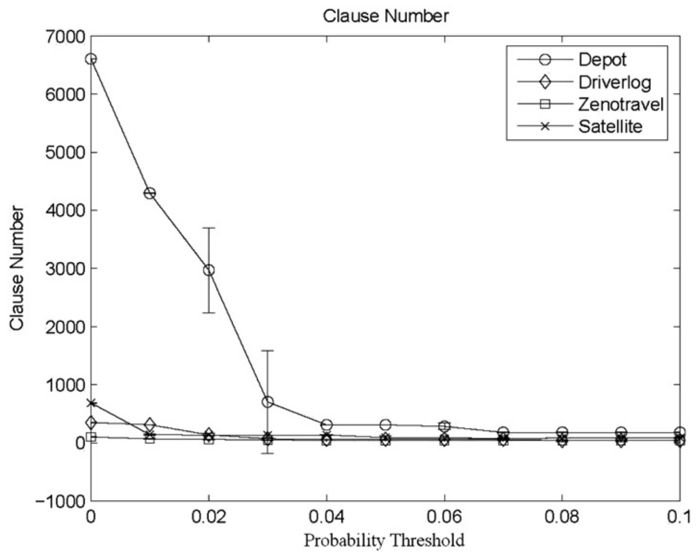

Fig. 4. Varying the probability threshold in four planning domains (Clause Number).

In all experiments, we wish to confirm the following hypotheses:

Hypothesis 1 `ARMS` can learn accurate action models with low error rates and redundancy rates for domains under different experimental conditions.
Hypothesis 2 `ARMS` can accomplish the learning task within a reasonable amount of computational time.

The first hypothesis will be verified by checking the average error rates E accumulated on the learned models when these models are applied to the testing examples, as well as the associated redundancy rates R. The second hypothesis will be verified through the average CPU time and the number of clauses incurred on the 160 training plan examples (we reserve 40 for testing). We use "$\pm$" to denote the range of the error bar for the error rate and redundancy rate as shown in subsequent tables.

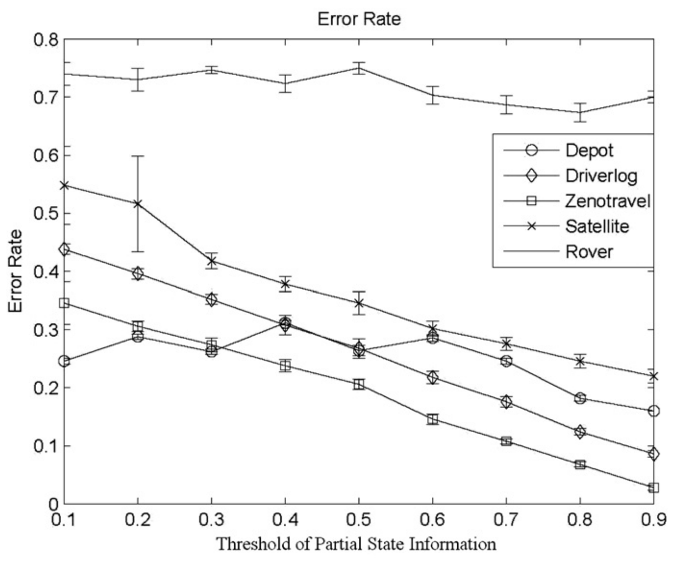

Fig. 5. Varying the partial state information (Error Rate).

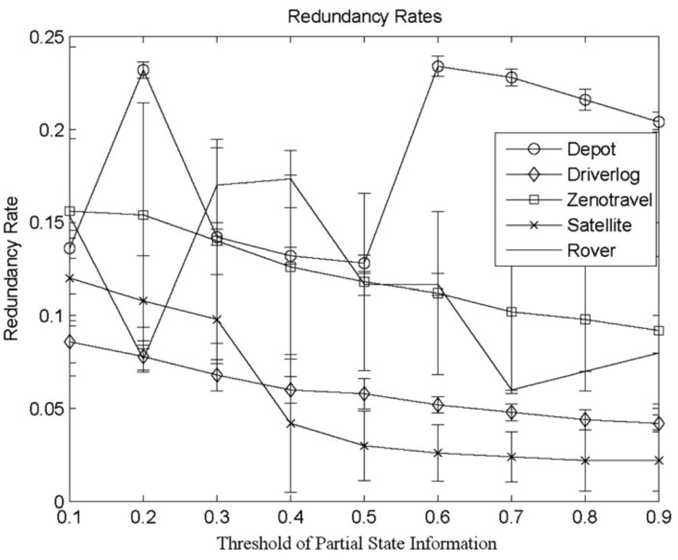

Fig. 6. Varying the partial state information (Redundancy Rate).

In all subsections except the last one, we use the weighted MAX-SAT solver tool [6] to implement ARMS. In the Section 6.3.4, we replace MaxSatSolver with the MaxWalkSat system and report the comparison results.

6.2. Learning in the depot domain

To give the readers an intuitive feeling of the resultant model learned by ARMS, we first describe the output generated by ARMS for the Depot and Driverlog domains. In these learned models, it is possible to visually compare the learned action models with the ground-truth 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 relation appears in both models, then it is shown by a *normal* font. If a relation appears only in M_i but not in M_l, then it is shown by an *italic* font. Otherwise, if a

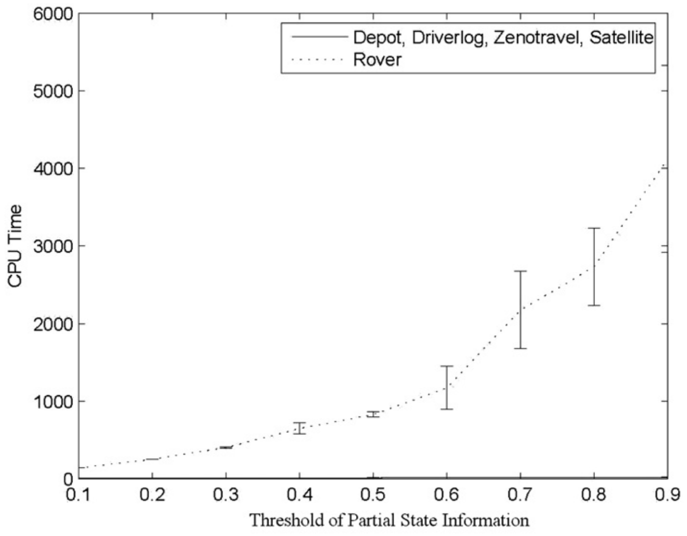

Fig. 7. Varying the partial state information (CPU Time).

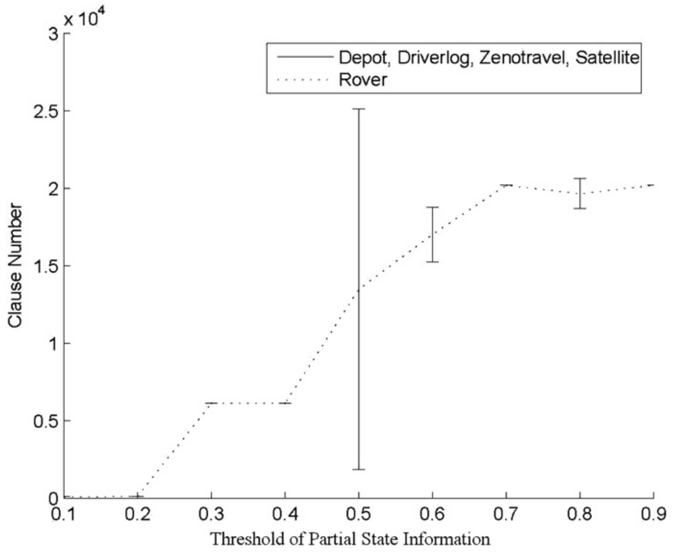

Fig. 8. Varying the partial state information (Clause Number).

relation will only appear in M_l and not in M_i, it is shown in **bold** font. As can be seen in Tables 9 and 10, most parts of the learned action models are correct with respect to the ground-truth domain descriptions.

6.3. *Varying independent variables*

We conducted a series of systematic evaluations of ARMS in the above-mentioned six PDDL-STRIPS domains from the IPC 2002 competition. In these experiments, we vary the three independent parameters and measure the error rate, redundancy rate, CPU time and robustness through cross-validation. In the subsequent experiments, E is the error rate, and R is the redundancy rate. CPU time is measured in seconds, and the values are the means of five-fold cross validation.

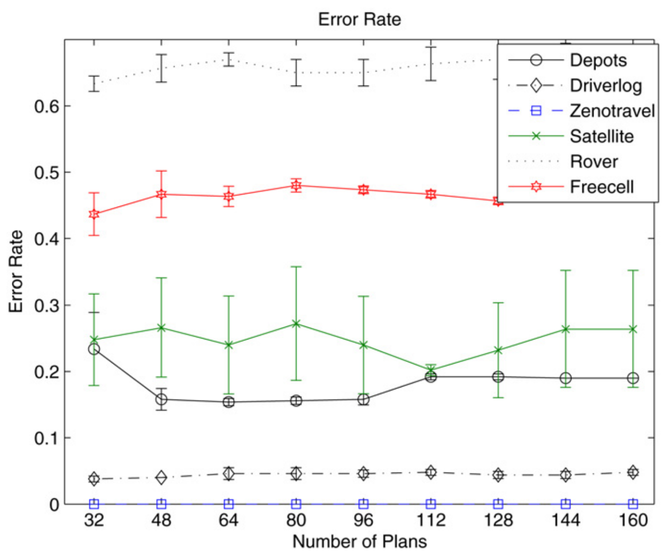

Fig. 9. Varying the number of plans (Error Rate).

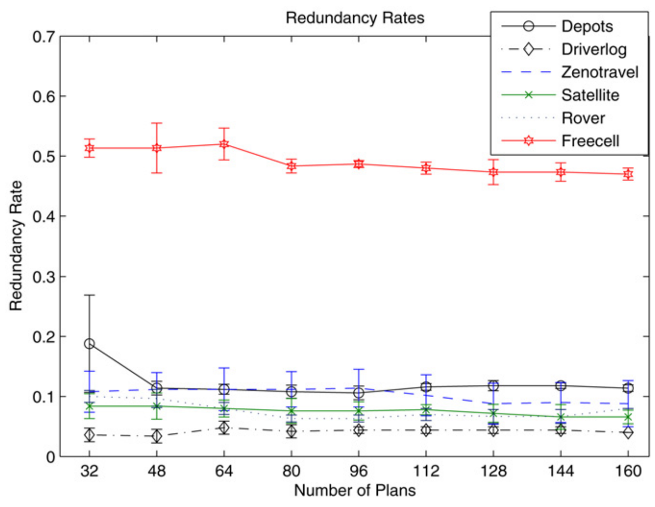

Fig. 10. Varying the number of plans (Redundancy Rate).

6.3.1. Varying the probability threshold

The probability threshold θ can be varied from 0% to 100%. When θ is high, fewer frequent action pairs remain for building the clauses in the weighted SAT formula, and as a result the computation becomes more efficient at a cost of potentially higher error and redundancy rates. In these experiments, the degree of partial information α is set to be zero (no observation of intermediate states in the middle of plans).

The results of varying θ in a range from zero to 100% in are given in Tables 11–16, and an excerpt of the data in a smaller range (from zero to 0.1%) in the table is shown graphically in Figs. 1–4 for ease of understanding.

We first focus on four of the six domains that show consistent performance in the entire probability threshold range from zero to 100%. As can be seen from Figs. 1–4, as we increase the probability threshold to 0.1%, the error rates of two of the domains, Depot and Satellite, generally increase with rather unstable behavior. In contrast, in the same range, the error rates of Zenotravel and Driverlog domains are quite stable and low. The redundancy rates for all four domains generally decrease in this smaller range, but stays stable over the larger range (Tables 11–16). On the CPU time and "Clause Number" charts, one can see that the efficiency of the system for all domains dramatically increases.

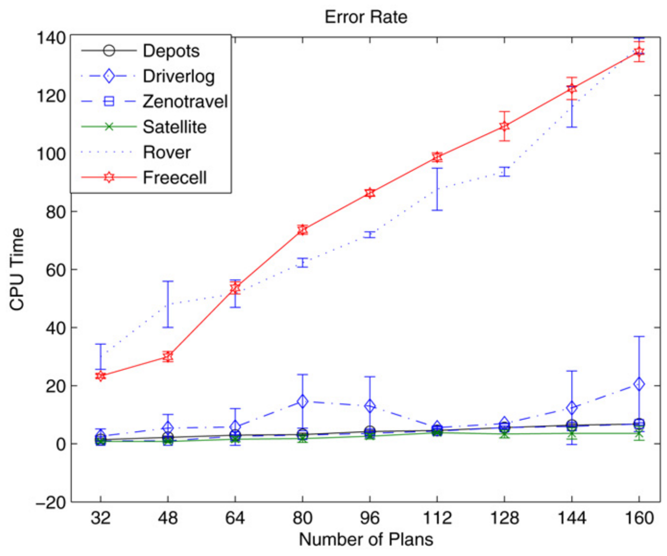

Fig. 11. Varying the number of plans (CPU Time).

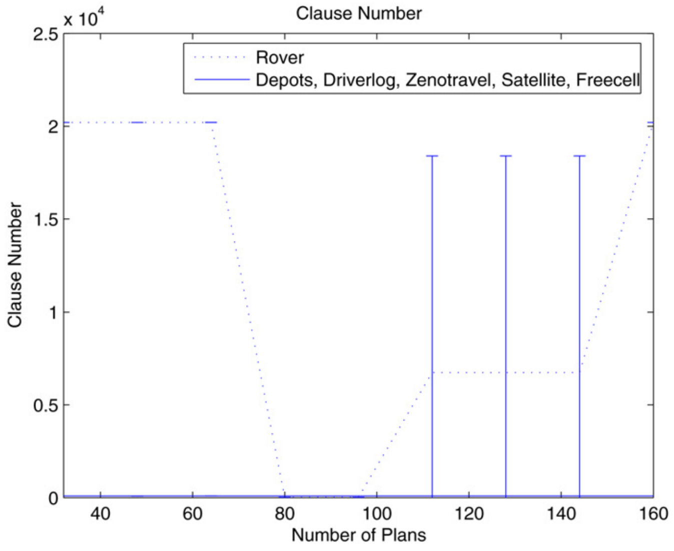

Fig. 12. Varying the number of plans (Clause Number).

Over the 100% range, we can see from Tables 11–16 that, as the probability threshold θ increases, the number of clauses decreases. At the same time, the CPU time decreases. This can be understood because when the probability threshold increases, fewer actions and relations are encoded into clauses. Thus, the computational work load also decreases. Two domains do not produce any result for θ below 60%, although they can give high quality action models when θ is above 60%. We can also see that when θ increases, the action models become slightly less accurate, which can also be seen from the increasing error rates. This is because the number of plan constraints decreases when the probability threshold θ increases, resulting in the ARMS system producing a simple action model when the number of plan constraints are small. Thus, the MaxSatSolver system becomes under-constrained. Overall, we can see that the error rate E and the redundancy rate R stay roughly at the same level when θ increases to 100% for all six domains. This indicates to us that the learned model is relatively stable while the efficiency can be improved greatly when we increase the probability threshold.

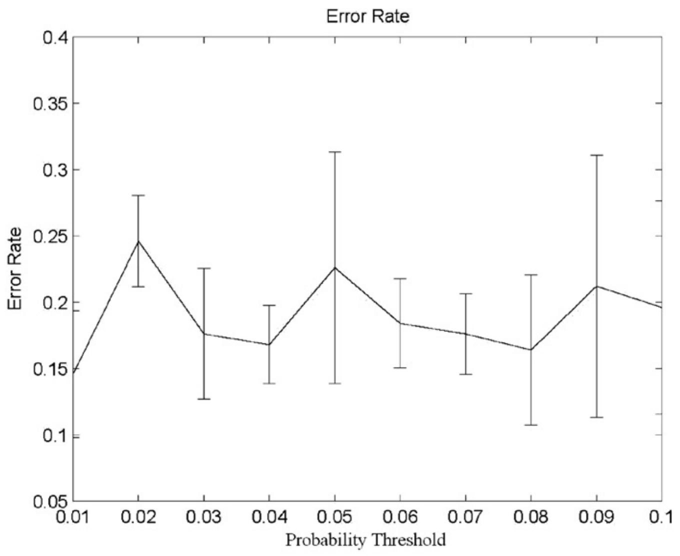

Fig. 13. Learning in the Depot domain using MaxWalkSat by varying probability threshold θ (Error Rate).

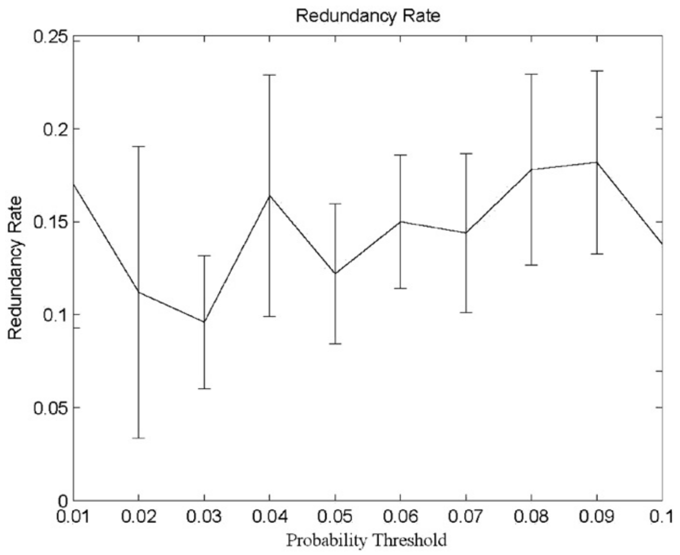

Fig. 14. Learning in the Depot domain using MaxWalkSat by varying probability threshold θ (Redundancy Rate).

For two of the six domains, Rover and Freecell, learning becomes too computationally expensive to run on our PC when the value of θ is less than 60%. Thus, we can only learn action models within our computational resources after the $\theta = 60\%$ threshold. This is because these two domains are more complex than the other four, as can be seen from the large number of relations and initial state relations in these domains (see Table 8). From Figs. 1–4, we can also see that the Depot domain has the highest error rates. This is due to the fact that in the Depot domain, almost any two actions can be paired up as a frequent action pair. As a result, the clause numbers are very large, resulting in inaccurate solutions being found for the weighted MAX-SAT problem. This is confirmed by the "clause number" chart in Figs. 1–4, where the average number of clauses for the Depot domain is the highest. This tells us that ARMS can perform well when the number of potential clauses is small and when the number of frequent pairs is also small.

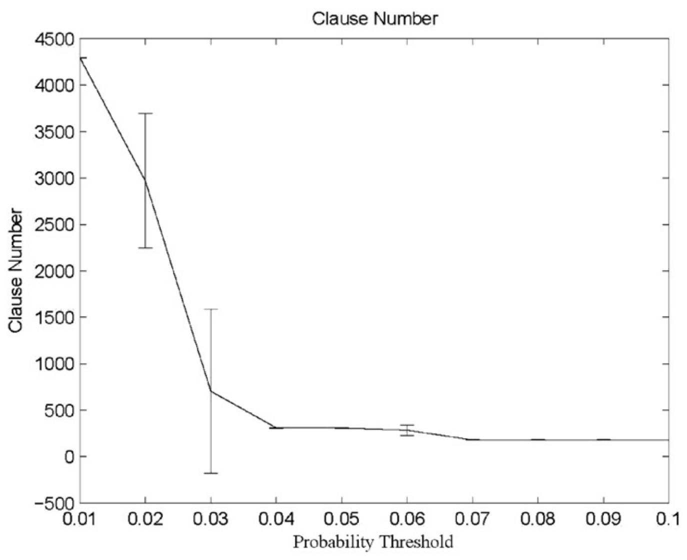

Fig. 15. Learning in the Depot domain using MaxWalkSat by varying probability threshold θ (CPU Time).

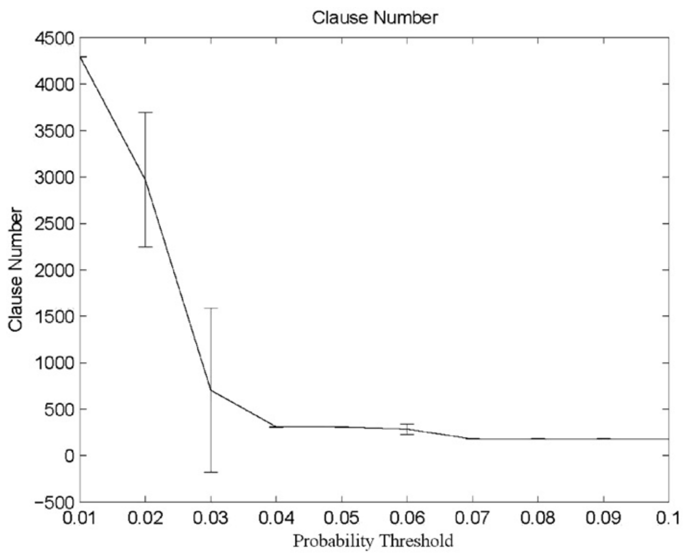

Fig. 16. Learning in the Depot domain using MaxWalkSat by varying probability threshold θ (Clause Number).

6.3.2. Varying the degree of partial information

As mentioned in the beginning of the section, the degree of partial information is designed to test how the ARMS algorithm might benefit from some partial information that can be occasionally observed in the middle of plans. This variable is denoted by p which is a value that ranges from zero to 100%. When p ranges from zero to 100%, the amount of available state information increases. The observed relations can then be used to explain frequent pairs of actions as additional clauses. These clauses that involve additional observations are given higher weights than other relations. Thus, as we increase p, we expect to see an increase in accuracy, as well as an increase in the number of clauses.

The experimental results are shown in Figs. 5–8. In all these experiments, we have fixed the probability threshold θ at 10%. As we can see, when more intermediate state information is known among the plans, the number of clauses becomes larger and the CPU time increases. However, knowing more information between the states makes the solution better, which can be seen from the error and redundancy rates. As we expected, the Freecell domain produces

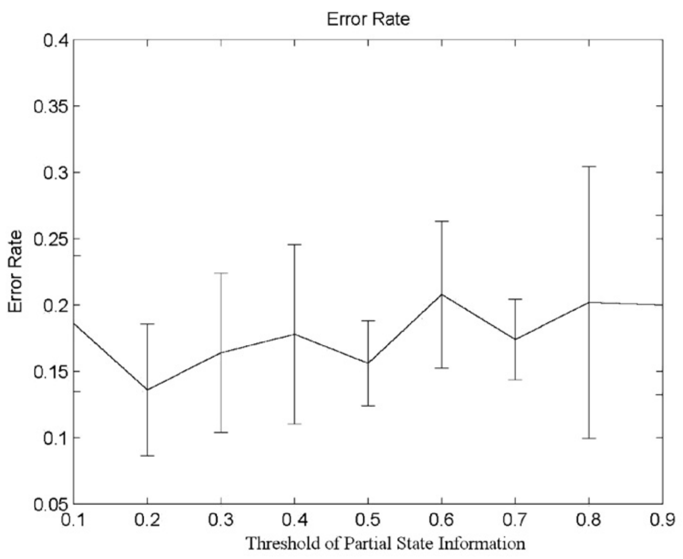

Fig. 17. Learning in the Depot domain using MaxWalkSat by varying the degree of partial information p (Error Rate).

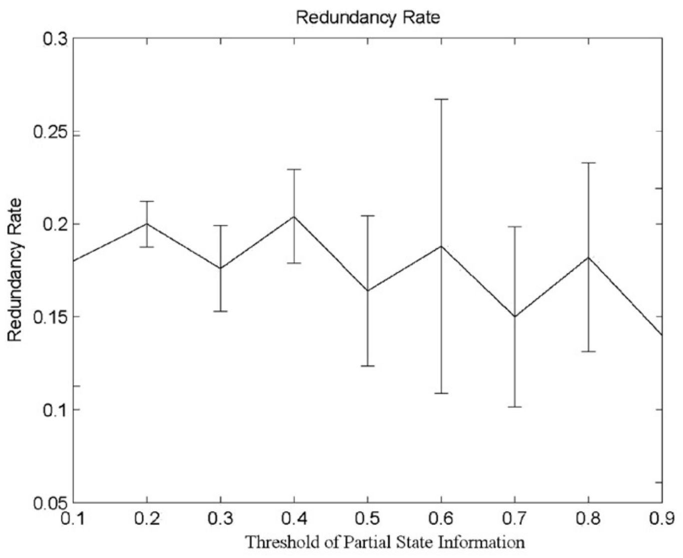

Fig. 18. Learning in the Depot domain using MaxWalkSat by varying the degree of partial information p (Redundancy Rate).

too many clauses for ARMS to handle due to its large number of relations, when we enforce the partial information constraints. Thus, no results are reported for this domain. We can conclude that when the number of relations are small, knowing more state information helps with accuracy and conciseness of the learned action model at a cost of more demanding computation.

6.3.3. *Varying the number of plans*

In order to test whether the size of the training set affects the quality of the learned model, we have conducted experiments that vary the number of plans in the training data set. In Figs. 9–12, we vary the number of plans in the training set from 32 to 160 and record the error and redundancy rates for the quality of the model. We also record the number of clauses and the CPU time. As we can see, as the training set increases its size, the error rate E and the redundancy rate R stay more or less at the same level. However, the CPU time and the number of clauses increase as

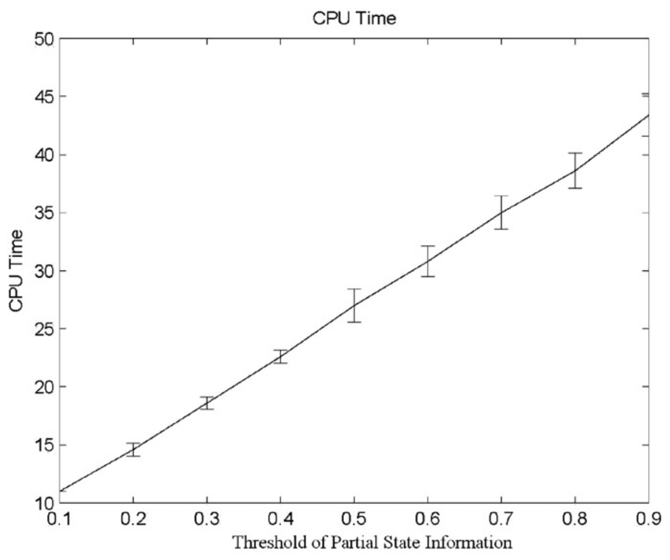

Fig. 19. Learning in the Depot domain using MaxWalkSat by varying the degree of partial information p (CPU Time).

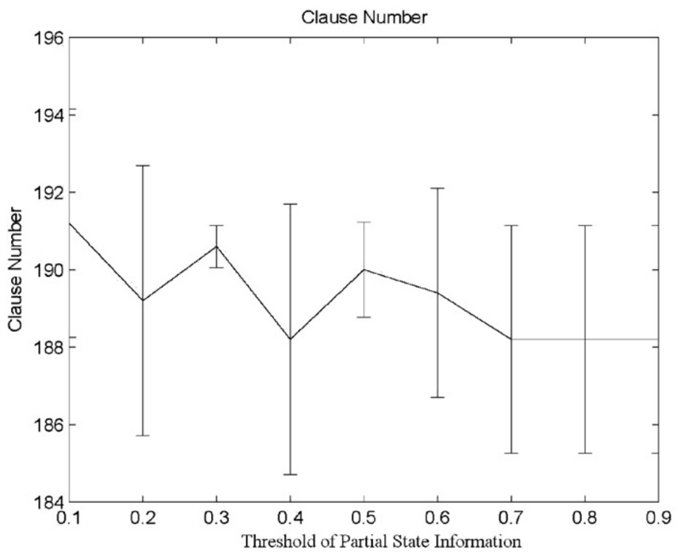

Fig. 20. Learning in the Depot domain using MaxWalkSat by varying the degree of partial information p (Clause Number).

expected. This shows that our system is quite stable in learning the action models even when the training set sizes are small.

6.3.4. Using MaxWalkSat as the weighted MAX-SAT solver

MaxWalkSat is another powerful solver for finding an approximate solution to weighted MAX-SAT problems [11,26]. It aims to minimize the weights associated with unsatisfied clauses, while local minima are avoided by randomly flipping the assignment of a variable in an unsatisfied clause [26]. By applying a different MAX-SAT solver, we hope to establish the fact that our learning algorithm is general in that it is independent of the actual tool that we select. As a local search algorithm, MaxWalkSat terminates when the sum of the weights of the un-satisfied clauses falls below a predefined target cost value. We have conducted a series of experiments where we replaced MaxSatSolver by MaxWalkSat in solving a weighted MAX-SAT problem. We are able to show similar learning performance as in MaxSatSolver experiments.

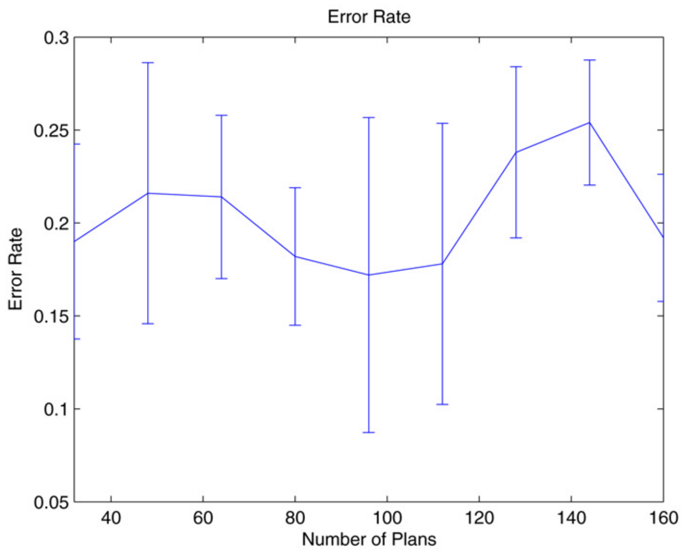

Fig. 21. Learning in the Depot domain using MaxWalkSat by varying the Number of Plans (Error Rate).

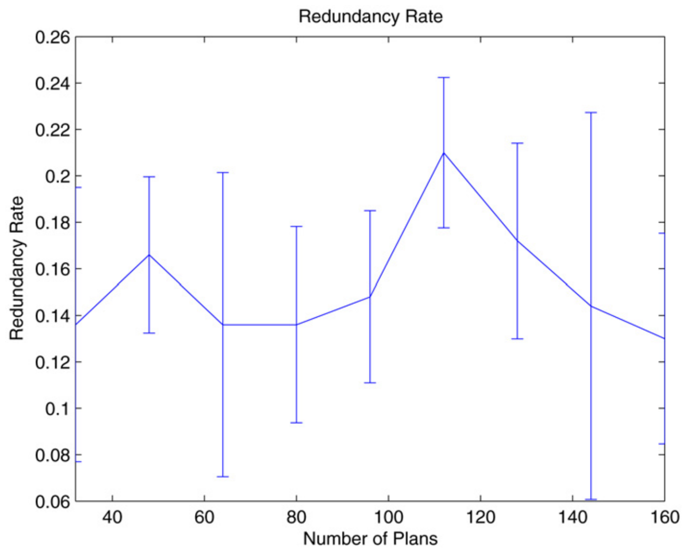

Fig. 22. Learning in the Depot domain using MaxWalkSat by varying the Number of Plans (Redundancy Rate).

We illustrate the results of using MaxWalkSat solver in ARMS. Figs. 13–16 plot the error rate, redundancy rate, CPU time and average number of clauses in the SAT formulae for the Depot domain. Figs. 17–20 show the error rate, redundancy rate, CPU time and average number of clauses as a function of the amount of partial information known in the intermediate states in a plan. Finally, Figs. 21–24 show the same metrics as a function of the number of plans in the training data set from the Depot domain. Comparing these figures and Figs. 1–12, we can see that the MaxWalkSat and MaxSatSolver performs similarly in terms of both the quality of the solution and the efficiency.

Finally, we compared the CPU times of MaxWalkSat and MaxSatSolver systems on the same planning problems, for both training and testing, for the Depot domain. The results are shown in Figs. 25–27. In this figure, the error bars are obtained from five fold cross validation experiments. As can be seen from the figures, the systems show similar performance and trends in solving the Depot domain learning problems.

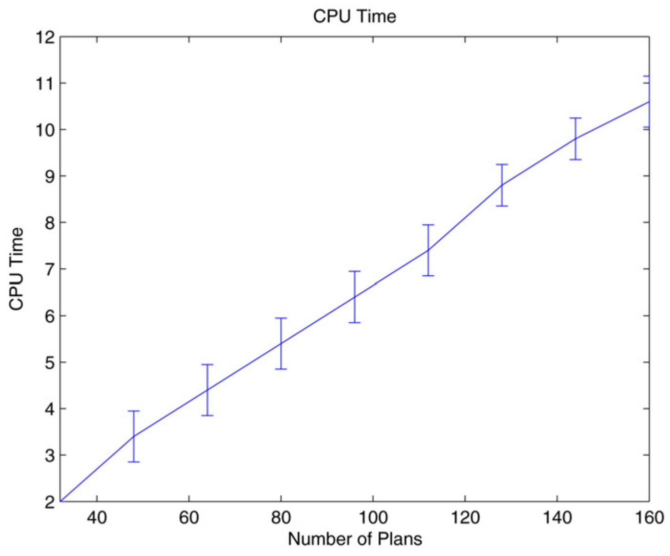

Fig. 23. Learning in the Depot domain using MaxWalkSat by varying the Number of Plans (CPU Time).

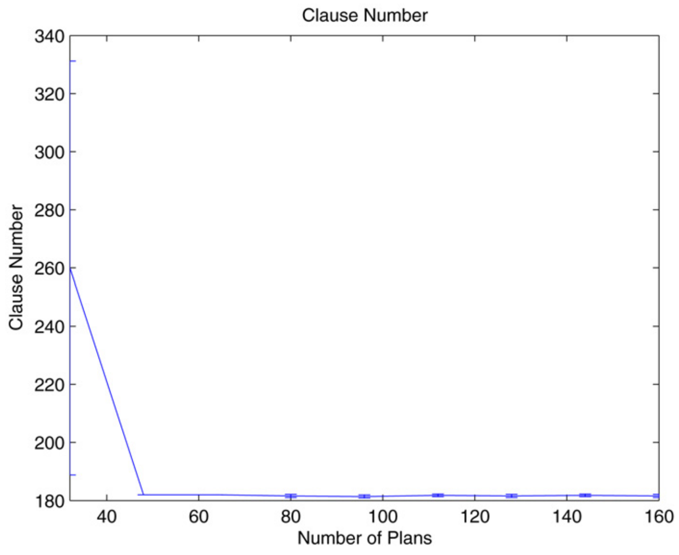

Fig. 24. Learning in the Depot domain using MaxWalkSat by varying the Number of Plans (Clause Number).

7. Conclusions and future work

In this paper, we have developed an algorithm for automatically learning action models from a set of plan examples where the intermediate states can be unknown. With a domain description and a set of successful plan traces, we can learn an action model that approximates the ideal models designed by human experts. We have shown empirically that it is feasible to learn these models in a reasonably efficient way using a weighted MAX-SAT solution even when the intermediate states are not observed, which removes a significant burden on the accumulation of the training data. Our learned action models can then be edited by experts before they are applied for practical planning.

While we are able to learn reasonable action models from plan traces, there are a few limitations of the current work, which we plan to overcome in our future extensions.

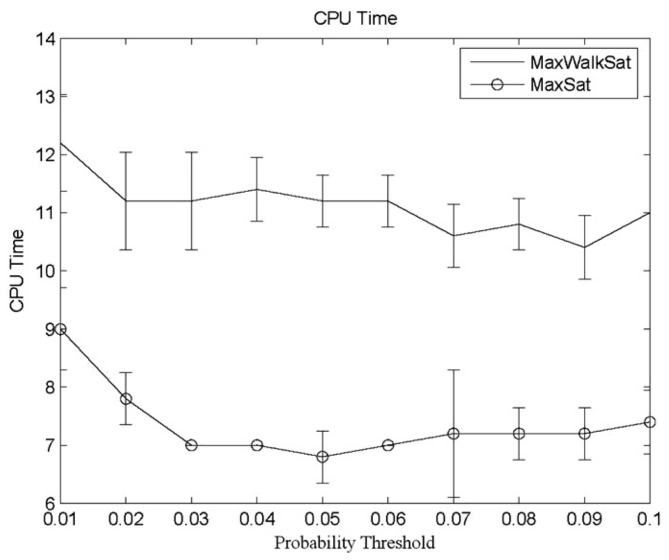

Fig. 25. A comparison of MaxWalkSat and MaxSat Solver (Probability Threshold).

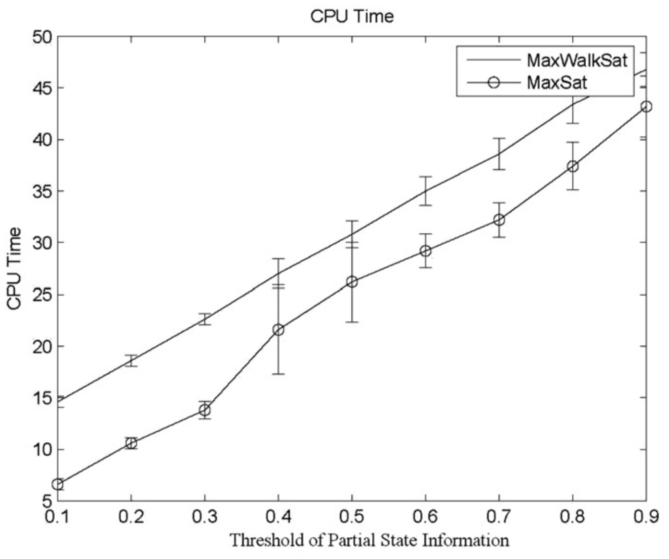

Fig. 26. A comparison of MaxWalkSat and MaxSat Solver (Threshold of Partial State Information).

(1) Our algorithm starts from an empty action model. However, in the real world, it is often the case that there is a partially completed existing action model to start with. It would be interesting to consider how to extend ARMS to allow a possibly imperfect or incomplete action model to be part of the input. We conjecture that in such situations, the existing action models can be used to bias the learning process, by simplifying the weighted MAX-SAT formulation, thus making the learning more effective.
(2) We have assumed that ARMS is given a correct set of plan traces as training data. However, the real world is full of noise, which exists in the form of incomplete observed action sequences or even incorrect observations of actions. Furthermore, some plans may fail to achieve some goals. Thus, it would be interesting to consider how to extend ARMS to handle noise in the training data.
(3) 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

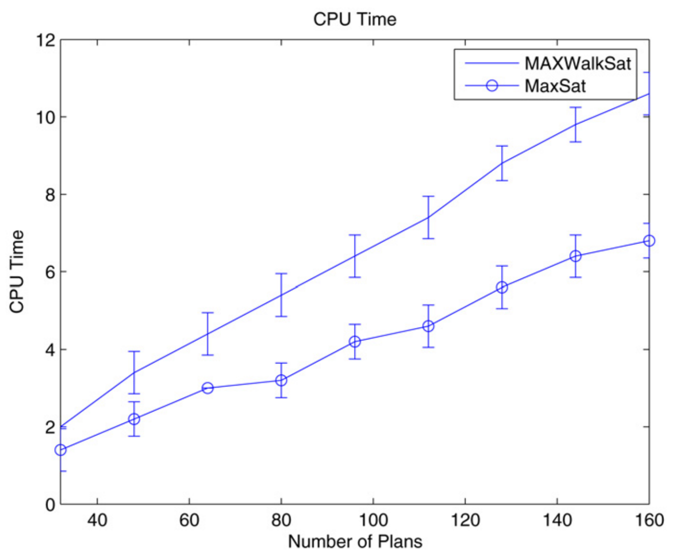

Fig. 27. A comparison of MaxWalkSat and MaxSat Solver (Number of Plans).

the generated plans using the learned models. We believe this can be accomplished by interactively repairing the models with the help of human experts.

(4) 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, 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.

(5) We wish to explore the application of ARMS *iteratively* on sets of actions in a collection of plans in the order of decreasing support measure, by following the planning with an abstraction approach. This allows the most frequent action sets to be explained first, thus causing fast convergence and producing superior action models.

(6) As we have pointed out in Section 5.4, there is a strong connection between ARMS algorithm, the action-model learning problem and the framework of Markov Logic Networks (MLN). We have pointed out how to model the action-model learning problem using the MLN by mapping the various components, including the constraints and weights, so that a solution to MLN corresponds to a learned action model. In the future, we wish to explore this connection further, especially on the issue of how to allow more expressive types of action models to be learned.

(7) Another possibility is to involve humans in the loop of the learning process. In the mix-initiative planning framework of Myers et al. [31], human users and computer systems work together in completing a planning task. While a system identifies possible candidates to be selected in completing a plan sketch, the human users make choices among the options. In our future work, we will consider how to involve human users in the learning process, so that when too many potential options exist for explaining frequent sets of actions in the training plans, human experts can be biased about making the final selection in order to obtain a high quality action model.

(8) In the related work section, we reviewed the CaMeL system [23], which acquires the method preconditions of a hierarchical task network (HTN) planning domain. We mentioned that in [23], plan traces as well as the derivational trees for these traces are taken as input into the learning system, which then produces a set of preconditions for HTN schemata. It would be interesting to consider two possible directions of future works. First, it would be interesting to include in the input of ARMS additional forms of domain theory such as the derivational traces into the learning process. With this additional knowledge, we would expect that the weighted MAX-SAT solution can be made more effective due to the additional bias in learning. A second direction is to extend ARMS to learning HTN method preconditions as well as the HTN schemata themselves. Such an extension and the comparison to the CaMel system would be an interesting future work.

(9) Finally, in the future we wish to evaluate best to determine the upper bound U of pre and post conditions for action models. This parameter can be experimentally found by varying U from a small value to a large value in

order to search for the lowest error and redundancy rates on test plans. The best such value can be determined empirically. We wish to continue to evaluate this strategy in our future work.

Acknowledgement

We wish to thank the Hong Kong RGC grant 621606 for supporting this research.

References

[1] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), Santiago de Chile, Chile, Morgan Kaufmann, 1994, pp. 487–499.
[2] E. Amir, Learning partially observable deterministic action models, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, UK, August 2005, pp. 1433–1439.
[3] M. Bain, C. Sammut, A framework for behavioural cloning, Machine Intelligence 15 (1996).
[4] S. Benson, Inductive learning of reactive action models, in: Proceedings of the International Conference on Machine Learning (ICML 1995), Stanford University, Stanford, CA, 1995, pp. 47–54.
[5] J. Blythe, J. Kim, S. Ramachandran, Y. Gil, An integrated environment for knowledge acquisition, in: Proceedings of the 2001 International Conference on Intelligent User Interfaces (IUI2001), Santa Fe, NM, 2001, pp. 13–20.
[6] B. Borchers, J. Furman, http://infohost.nmt.edu/~borchers/maxsat.html.
[7] B. Borchers, J. Furman, A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems, Journal of Combinatorial Optimization 2 (4) (1999) 299–306.
[8] M. Davis, H. Putnam, A computing procedure for quantification theory, Journal of The Association for Computing Machinery 7 (1960) 201–215.
[9] P. Domingos, S. Kok, H. Poon, M. Richardson, P. Singla, Unifying logical and statistical AI, in: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006), Boston, MA, July 2006.
[10] K. Erol, J.A. Hendler, D.S. Nau, Htn planning: Complexity and expressivity, in: Proceedings of the National Conference on Artificial Intelligence (AAAI 1994), Seattle, WA, 1994, pp. 1123–1128.
[11] H. Kautz, et al., http://www.cs.washington.edu/homes/kautz/walksat/.
[12] W.H. Evans, J.C. Ballegeer, N.H. Duyet, ADL: An algorithmic design language for integrated circuit synthesis, in: Proceedings of the 21st Design Automation Conference on Design Automation, 1984.
[13] R.E. Fikes, N.J. Nilsson, Strips: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (1971) 189–208.
[14] M. Fox, D. Long, PDDL2.1: An extension to pddl for expressing temporal planning domains, Journal of Artificial Intelligence Research 20 (2003) 61–124.
[15] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, 1979.
[16] A. Garland, N. Lesh, Plan evaluation with incomplete action descriptions, in: Proceedings of the Eighteenth National Conference on AI (AAAI 2002), Edmonton, Alberta, Canada, 2002, pp. 461–467.
[17] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL—the planning domain definition language, 1998.
[18] Y. Gil, Learning by experimentation: Incremental refinement of incomplete planning domains, in: Eleventh Intl. Conf. on Machine Learning, 1994, pp. 87–95.
[19] M.X. Goemans, D.P. Williamson, Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming, Journal of the ACM 42 (1995) 1115–1145.
[20] P.D. Grunwald, I.J. Myung, M.A. Pitt, Advances in Minimum Description Length Theory and Applications, MIT Press, Cambridge, MA, 2005.
[21] O. Ilghami, H. Munoz-Avila, D.S. Nau, D.W. Aha, Learning preconditions for planning from plan traces and HTN structure, Journal of Artificial Intelligence Research 20 (2003) 379–404.
[22] O. Ilghami, H. Munoz-Avila, D.S. Nau, D.W. Aha, Learning preconditions for planning from plan traces and htn structure, in: Proceedings of the International Conference on Machine Learning (ICML 2005), Bonn, Germany, 2005.
[23] O. Ilghami, D.S. Nau, H. Munoz-Avila, Camel: Learning method preconditions for htn planning, in: Proceedings of the Sixth International Conference on AI Planning and Scheduling AIPS-02, Toulouse, France, 2002, pp. 168–178.
[24] R.G. Jeroslow, J. Wang, Solving propositional satisfiability problems, Annals of Mathematics and AI 1 (1990) 167–187.
[25] H. Kautz, B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996), Portland, OR, 1996, pp. 1194–1201.
[26] H. Kautz, B. Selman, Y. Jiang, A general stochastic approach to solving problems with hard and soft constraints, in: The Satisfiability Problem: Theory and Applications, 1997.
[27] T. Lau, P. Domingos, D.S. Weld, Version space algebra and its application to programming by demonstration, in: Proc. 17th International Conf. on Machine Learning, Morgan Kaufmann, San Francisco, CA, 2000, pp. 527–534.
[28] D.W. Loveland, Automated Theorem Proving: A Logical Basis, North-Holland, New York, 1978.
[29] T. Leo McCluskey, D. Liu, R.M. Simpson, GIPO II: HTN planning in a tool-supported knowledge engineering environment, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2003), Trento, Italy, 2003, pp. 92–101.

[30] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient sat solver, in: Proceedings of the 38th Design Automation Conference (DAC), 2001.
[31] K.L. Myers, P. Jarvis, W.M. Tyson, M. Wolverton, Mixed-initiative framework for robust plan sketching, in: Thirteenth International Conference on Automated Planning and Scheduling (ICAPS-03), BC, Canada, AAAI Press, 2003, pp. 256–266.
[32] D.S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu, F. Yaman, Shop2: An HTN planning system, Journal of Artificial Intelligence Research 20 (2003) 379–404.
[33] D.S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu, F. Yaman, Applications of shop and shop2, IEEE Intelligent Systems 20 (2) (2005) 34–41.
[34] T. Oates, P.R. Cohen, Searching for planning operators with context-dependent and probabilistic effects, in: Proceedings of the Thirteenth National Conference on AI (AAAI 1996), Portland, OR, 1996, pp. 865–868.
[35] M. Richardson, P. Domingos, Markov logic networks, Technical Report, 2004.
[36] M. Richardson, P. Domingos, Markov logic networks, Machine Learning 62 (1–2) (July 2006) 107–136.
[37] G. Sablon, D. Boulanger, Using the event calculus to integrate planning and learning in an intelligent autonomous agent, in: Current Trends in AI Planning, IOS Press, 1994, pp. 254–265.
[38] B. Selman, H. Kautz, An empirical study of greedy local search for satisfiability testing, in: Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93), Washington, DC, 1993, pp. 46–51.
[39] B. Selman, H. Kautz, B. Cohen, Local search strategies for satisfiability testing, Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge 26 (October 11–13, 1993).
[40] D. Shahaf, E. Amir, Learning partially observable action schemas, in: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006), Boston, MA, July 2006.
[41] D. Shahaf, A. Chang, E. Amir, Learning partially observable action models: Efficient algorithms, in: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006), Boston, MA, July 2006.
[42] W. Shen, Autonomous Learning from the Environment, Computer Science Press/W.H. Freeman and Company, 1994.
[43] X. Wang, Learning by observation and practice: An incremental approach for planning operator acquisition, in: Proceedings of the Twelfth International Conference on Machine Learning (ICML 1995), 1995, pp. 549–557.
[44] E. Winner, M. Veloso, Analyzing plans with conditional effects, in: Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS 2002), Toulouse, France, 2002.
[45] Q. Yang, Formalizing planning knowledge for hierarchical planning, Computational Intelligence Journal 6 (2) (1990) 12–24.
[46] Q. Yang, K. Wu, Y. Jiang, Learning action models from plan examples with incomplete knowledge, in: Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), Monterey, CA, 2005, pp. 241–250.
[47] J. Yin, D. Shen, Q. Yang, Z.-N. Li, Activity recognition through goal-based segmentation, in: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05), Pittsburgh, PA, 2005, pp. 28–34.
[48] J. Yin, Q. Yang, Integrating hidden Markov models and spectral analysis for sensory time series clustering, in: Proceedings of the Fifth IEEE International Conference on Data Mining, Houston, TX, 2005.
[49] H. Zhang, SATO: An efficient propositional prover, in: Proceedings of the 14th International Conference on Automated Deduction (CADE 1997), Townsville, North Queensland, Australia, 1997, pp. 272–275.