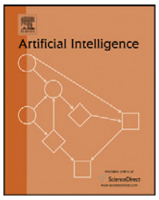
Artificial Intelligence 174 (2010) 1540–1569

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



Learning complex action models with quantifiers and logical implications

Hankz Hankui Zhuo[a,b], Qiang Yang[b,*], Derek Hao Hu[b], Lei Li[a]

[a] *Department of Computer Science, Sun Yat-sen University, Guangzhou, China, 510275*
[b] *Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clearwater Bay, Kowloon, Hong Kong*

ARTICLE INFO

Article history:
Received 2 January 2010
Received in revised form 4 September 2010
Accepted 5 September 2010
Available online 29 September 2010

Keywords:
Action model learning
Machine learning
Knowledge engineering
Automated planning

ABSTRACT

Automated planning requires action models described using languages such as the **P**lanning **D**omain **D**efinition **L**anguage (PDDL) as input, but building action models from scratch is a very difficult and time-consuming task, even for experts. This is because it is difficult to formally describe all conditions and changes, reflected in the preconditions and effects of action models. In the past, there have been algorithms that can automatically learn simple action models from plan traces. However, there are many cases in the real world where we need more complicated expressions based on universal and existential quantifiers, as well as logical implications in action models to precisely describe the underlying mechanisms of the actions. Such complex action models cannot be learned using many previous algorithms. In this article, we present a novel algorithm called LAMP (**L**earning **A**ction **M**odels from **P**lan traces), to learn action models with quantifiers and logical implications from a set of observed plan traces with only partially observed intermediate state information. The LAMP algorithm generates candidate formulas that are passed to a Markov Logic Network (MLN) for selecting the most likely subsets of candidate formulas. The selected subset of formulas is then transformed into learned action models, which can then be tweaked by domain experts to arrive at the final models. We evaluate our approach in four planning domains to demonstrate that LAMP is effective in learning complex action models. We also analyze the human effort saved by using LAMP in helping to create action models through a user study. Finally, we apply LAMP to a real-world application domain for *software requirement engineering* to help the engineers acquire software requirements and show that LAMP can indeed help experts a great deal in real-world knowledge-engineering applications.

1. Introduction

Automated planning systems achieve goals by producing sequences of actions from the given action models that are provided as input [14]. A typical way to describe the action models is to use action languages such as the **P**lanning **D**omain **D**efinition **L**anguage (PDDL) [13,11,14] in which one can specify the precedence and consequence of actions. A traditional way of building action models is to ask domain experts to analyze a task domain and manually construct a domain description that includes a set of complete action models. Planning systems can then proceed to generate action sequences to achieve goals.

However, it is very difficult and time-consuming to manually build action models in a given task domain, even for experts. This is a typical problem of the knowledge-engineering bottleneck, where experts often find it difficult to articulate their experiences formally and completely. Because of this, researchers have started to explore ways to reduce the human

* Corresponding author.
E-mail addresses: zhuohank@mail.sysu.edu.cn (H.H. Zhuo), qyang@cse.ust.hk (Q. Yang), derekhh@cse.ust.hk (D.H. Hu), lnslilei@mail.sysu.edu.cn (L. Li).

0004-3702/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2010.09.007

effort of building action models by learning from observed examples or plan traces. Some researchers have developed methods to learn action models from complete state information before and after an action in some example plan traces [4, 15,31,50]. Others, such as Yang et al. [51,39] have proposed to learn action models from plan examples with only incomplete state information. Yang et al. [51,52] have developed an approach known as *Action Relation Modeling System* (ARMS) to learn action models in a STRIPS (**ST**anford **R**esearch **I**nstitute **P**roblem **S**olver) [10] representation using a weighted MAXSAT-based (Maximum Satisfiability) approach. Shahaf et al. [39] have proposed an algorithm called *Simultaneous Learning and Filtering* (SLAF) to learn more expressive action models using consistency-based algorithms.

Despite the success of these learning systems, in the real world, there are many applications where actions should be expressed using a more expressive representation, namely, quantifiers and logical implications. For instance, consider the case where there are different *cases* in a *briefcase*[1] planning domain, such that a briefcase should not be moved to a place where there is another briefcase with the same color. We can model the action *move* in PDDL as follows.[2]

```
action:  move(?c1 - case ?l1 ?l2 - location)
pre:     (:and (at ?c1 ?l1)
           (forall ?c2 - case (imply (samecolor ?c2 ?c1)(not (at ?c2 ?l2)))))
effect:  (:and (at ?c1 ?l2) (not (at ?c1 ?l1)))
```

That is, if we want to move the case *c1* from the location *l1* to *l2*, *c1* should be at *l1* first, and every other case *c2* whose color is the same with *c1* should not be at *l2*. After the action *move*, *c1* will be at *l2* instead of at *l1*. Likewise, consider a pilot could not fly to a place where there are enemies. We can model the action model *fly* as follows.

```
action:  fly(?p1 - pilot ?l1 ?l2 - location)
pre:     (:and (at ?p1 ?l1)
           (forall ?p2 - person (imply (enemy ?p2 ?p1)(not (at ?p2 ?l2)))))
effect:  (:and (at ?p1 ?l2) (not (at ?p1 ?l1)))
```

We can see that in these examples, we need universal quantifiers as well as logical implications in the precondition part of the action to precisely represent this action and compress the action model in a compact form.

As another example, consider a driver who intends to drive a train. Before he can start, he should make sure all the passengers have gotten on the train. After that, if there is a seat vacant, then he can start to drive the train. We represent this *drive-train* action model in PDDL as follows.

```
action:  drive-train(?d - driver ?t - train)
pre:     (free ?d) (forall ?p - passenger (in ?p ?t))
effect:  (:and (when (exist ?s - seat (vacant ?s)) (available ?t))
               (driving ?d ?t)(not (free ?d)))
```

That is, if a driver *?d* makes sure all the passengers *?p* are in the train *?t* and is free at that time, then he can drive the train *?t*. Furthermore, if there is a seat *?s* vacant, as a consequence of this action *drive-train*, the train will be set as available to show that more passengers can take this train. Besides, the driver *?d* will be in the state of driving the train, i.e., *(driving ?d ?t)*, and not free. Such an action model needs a universal quantifier in describing its preconditions and an existential quantifier for the condition "*(exist ?s - seat (vacant ?s))*" of the conditional effect "*(when (exist ?s - seat (vacant ?s))(available ?t))*". More examples that require the use of quantifiers and logical implications can be found in many action models in recent International Planning Competitions, such as the domains in IPC-5[3]: *trucks*, *openstacks*, etc. These complex action models can be represented by PDDL, but cannot be learned by existing algorithms proposed for action model learning.

Our objective is to develop a new algorithm for learning complex action models with quantifiers (including conditional effects) and logical implications, from a collection of given example plan traces. The input of our algorithm includes: (1) a set of observed plan traces with partially observed intermediate state information between actions; (2) a list of action headings, each of which is composed of an action name and a list of parameters, but is not provided with preconditions or effects; (3) a list of predicates along with their corresponding parameters. Our algorithm is called LAMP (***L**earn **A**ction **M**odels from **P**lan traces*), which outputs a set of action models with quantifiers and implications. These action models 'summarizes' the plan traces as much as possible, and can be used by domain experts, who need to spend only a small amount of time, in revising parts of the action models that are incorrect or incomplete, before finalizing the action models for planning usage.

Compared to many previous approaches, our main contributions are: (1) LAMP can learn quantifiers that conform to the PDDL definition [13,11], where the latter article shows that action models in PDDL can have quantifiers. (2) LAMP can learn action models with implications as preconditions, which improves the expressiveness of learned action models. We require

[1] http://www.informatik.uni-freiburg.de/~koehler/ipp/pddl-domains.tar.gz.

[2] A symbol with a prefix "?" suggests that the symbol is a variable; e.g. "?c1" suggests that "c1" is a variable that can take on certain constants as values.

[3] http://zeus.ing.unibs.it/ipc-5/domains.html.

that existential quantifiers can only be used to quantify the left-hand side of a conditional effect, which is consistent with the constraints used in PDDL1.2 and PDDL2.1 [13,11]. (3) Similar to some previous systems such as the ARMS system [51, 52], LAMP can learn action models from plan traces with *partially observed* intermediate state information. This is important because in many real world situations, what we can record between two actions in a plan trace is likely to be incomplete; e.g., when using only a small number of sensors, we can record a subset of what happens after each action is executed. In such a case, the number of sensors cannot cover all possible new information sources and events after each action. Therefore, the state information we observe is only incomplete.

One might further ask the question of whether a large number of plan traces could be provided when the full description of action models are unavailable. Here we provide several examples to demonstrate the validity of such requirements in some real-world applications. One example is batch commands in an operating system that consist of the name of each command along with some partial information about directory location, structure and content. If we view such an application as a planning application, a batch command is a list of actions. We can easily get the action headings (action names and its parameters) by using the UNIX history command. However, we cannot get the full intermediate state information between these commands by reading the batch command file alone. This corresponds to an example of plan traces with partial intermediate state information. In this domain, we do not have the full specification of action models, but we can easily get a large number of batch commands, viewed as plan examples, together with partial state information.

Another example is activity recognition [56], which is an active research area that integrates pervasive computing, machine learning and wireless sensor networks. Activity recognition aims to identify actions and goals of one or more agents from a series of observations on the agents' actions and environmental conditions. One scenario is sensor-based activity recognition [60,19], where we can use the sensor readings in a pervasive computing environment to understand what activities are being carried out by collecting a large number of sensor reading sequences. We can then perform activity recognition by mapping these sensor-reading sequences to the corresponding activity sequences. These activity sequences can then act as the input to action-model learning algorithms to obtain action models, which can further improve the accuracy of activity recognition. However, due to the uncertainty of sensor readings, we may only have partial knowledge about the activities learned. Thus, the state information in the activity sequences learned may sometimes be only partially observable. Therefore, in an activity-recognition domain, we may have a large number of plan sequences with partially observed intermediate state information, without knowing about what the model of each action is.

A third example comes from Web services, where many planning researchers have attempted to automate the process of linking Web services to achieve complex tasks [17]. Standard description languages such as SOAP [57] (Simple Object Access Protocol) are used to describe both the syntax and semantics of the services. However, although the standard protocols such as SOAP can describe the syntax of the Web services, it is rather difficult if we rely on Web-service providers to label the semantic content of each service, especially when we consider the case that the Web services may come from many different sources. However, it is easy to check the log data and acquire a large number of examples, which can then be used to learn the Web service behaviors from examples [42]. Similarly, in this scenario, we can get a large number of logs as plan traces, and use them to learn the Web service behaviors or action models, even when we do not have precise descriptions of all events happening after a service is executed.

We now give an overview of the LAMP algorithm. At the high level, the LAMP algorithm can be described in four steps. Firstly, we encode the input plan traces, including observed states and actions (represented as state transitions), into propositional formulas. Secondly, we generate candidate formulas according to the predicate lists and domain constraints. Thirdly, we build a Markov Logic Network (MLN) by learning the corresponding weights of formulas to select the most likely subset from the set of candidate formulas. Finally, we convert this subset into the final action models.

We conduct experiments on four planning domains to show that LAMP is effective in learning action models with quantifiers and logical implications. These action models may still be incorrect and incomplete. Thus, we define a quality measure for the learned action models. We show that these learned action models are very close to the hand-crafted action models that support quantifiers and logical implication. Furthermore, by conducting user studies, we demonstrate that human experts only need to spend a relatively small amount of time revising these learned action models, compared to writing these action models from scratch. Finally, we apply LAMP to a software requirement engineering domain to illustrate real-world applications of the proposed method.

The rest of the paper is organized as follows. We will first discuss some related works in the field of action model learning. In particular, we will discuss ARMS and SLAF in detail. Since this paper is related to Markov Logic Networks (MLNs), we will also review some papers on both the theoretical foundations and applications of MLNs. Next, we will give the definition of our problem and describe the detailed steps of our LAMP algorithm. We then evaluate our algorithm in four planning domains to learn the action models and show some desirable properties of these learned action models. Finally, we will conclude paper and discuss our future works.

2. Related work

2.1. Learning with full intermediate state information

Recently, some methods have been proposed to learn action models from plan traces automatically. The first one is to learn action models from plan traces with full intermediate state information [6,15,31,50,37,32,33]. Gil et al. [15] describe a system called EXPO, bootstrapped by an incomplete STRIPS-like domain description with the rest being filled in through experience. Oates et al. [31] use a general classification system to learn the effects and preconditions of actions, identifying irrelevant variables. Schmill et al. [37] try to learn operators with approximate computation in relevant domains by assuming that the world is fully observable. Wang [50] describes an approach to automatically learn planning operators by observing expert solution traces and refine the operators through practice in a learning-by-doing paradigm. It uses the knowledge naturally observable when experts solve problems. Chrisman [6] shows how to learn stochastic actions with no conditional effects. In [32,33], a probabilistic, relational planning rule representation can be learned which could compactly model noisy, non-deterministic action effects. Holmes et al. [18] model synthetic items based on experience to construct action models. Walsh and Littman [42] propose an efficient algorithm for learning action schemas for describing Web services. Among these methods, one of their limitations is all the intermediate observations need to be known. However, in many real applications such as activity recognition from wireless sensor networks, biological applications of AI Planning, intelligent user interfaces and Web services [14,24,16,12], sometimes we cannot obtain full intermediate state information.

2.2. Learning with partial intermediate state information

In the past, solutions have been developed to learn action models where the intermediate state information is not fully observable. These are the partially observable cases. In this area, two important algorithms are ARMS and SLAF.

ARMS [51,52] can automatically discover action models from a set of successfully observed plan traces. Unlike previous work in action model learning, it does not assume complete knowledge of states in the middle of observed plan traces. ARMS works when partial intermediate states are given. These plan traces are obtained by an observation agent who does not know the logical encoding of the actions and the full state information between the actions. Specifically, ARMS gathers knowledge from the statistical distribution of frequent sets of actions in plan traces. It builds a weighted propositional satisfiability problem and solves it using a weighted MAX-SAT solver. It extracts constraints from plan traces and STRIPS models by itself, and further deals with these constraints with weighted MAXSAT [5]. Finally, it attains STRIPS models from the output of weighted MAXSAT. ARMS can handle cases when intermediate observations are difficult to acquire, but it cannot learn action models with quantifiers or implications, which are more complex than STRIPS models.

While ARMS can learn STRIPS actions from plan traces, it is not designed to learn complex action models. The preconditions (or effects) of a STRIPS action model are literals, which makes it easy to build *action/plan constraints* (on PRE/ADD/DEL lists) and assign weights to them, as what was done in ARMS. However, the formulas can contain universal or existential quantifiers, which make the number of literals affected by these formulas unfixed and can be extremely large. One natural example is the way we handle wildcard characters in an operating system. Such wildcard characters like "*.*" can correspond to all files under a directory, which makes learning especially difficult and slow. Thus, in contrast to ARMS, we choose to build our LAMP algorithm based on a more natural approach for learning first-order logic formulas using a Markov Logic Network (MLN) [23].

Amir et al. [1,39,38,30] present a tractable, exact system SLAF for the problem of identifying actions' effects in partially observable STRIPS domains. It resembles version spaces and logical filtering and identifies all the models that are consistent with observations. It maintains and outputs a relational logical representation of all possible action-schema models after a sequence of executed actions and partial observations. To improve the performance, Shahaf et al. [39] propose an efficient algorithm to learn preconditions and effects of deterministic action models. For the quantifiers, Shahaf and Amir [38] impose constraints on its learning result so that the existential quantifiers must appear in preconditions and universal quantifiers appear in effects. In many real-world planning applications, however, existential quantifiers may appear in the effects of actions and universal quantifiers may appear in the preconditions of some actions, as shown in the examples of "*move*" and "*graduate*" in Section 1. Besides, it cannot learn action models with preconditions that include the form of implications either, which requires more complex action models in the real world. These action models, as noted in Section 1, are needed to model agents' complex behaviors in many real-world applications.

Cresswell et al. [44] present a system called *LOCM* designed to carry out automated induction of action models from sets of example plans. *LOCM* assumes that a planning domain consists of sets (called *sorts*) of object instances, where each object behaves in the same way as any other object in its sort [45]. Compared with previous systems, *LOCM* can learn action models with action sequences as input, and is shown to work well under the assumption that the output domain model can be represented in an object-centered representation.

Our previous work [53,54] established the feasibility of learning action models with conditional effects by transferring knowledge from other planning domains. In [55], we presented an algorithm to learn action models in STRIPS descriptions and Hierarchical Task Network (HTN) [59] method-preconditions simultaneously. In this paper, we illustrate in further detail our action-model learning system LAMP, which can learn complex action models including quantifiers, logic implications, as well as conditional effects. We also present an application example in software engineering [58] to show some real benefit gained by using the LAMP system.

2.3. Relation to knowledge acquisition and inductive logic programming

In [4], action models are acquired from a human expert via interactions. Simpson et al. [49,46] describe a Graphical Interface for Planning with Objects (called GIPO) that has been built to investigate and support the knowledge engineering process in the building of applied AI Planning systems. Winner and Veloso [47] present a DISTILL system to learn program-like plan templates from example plans, whose aim is to automatically acquire plan templates from example plans, where the templates can be used for planning directly. The focus of [47] is on how to extract plan templates from example plans as a substitute for planners. The plan templates represent the structural relations among actions. Another related work is [48] where a programming by demonstration (PBD) problem is solved using a version-space learning algorithm by acquiring 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. Different from our work, the aim of the PBD system [48] is to learn action sequences instead of action models.

Another related work is [36], where Sablon and Bruynooghe present a method to learn action models from its own experience and from its observation of a domain expert. It exploits the idea of concept induction in first-order predicate logic of inductive logic programming (ILP) [29], which allows it to utilize ILP noise-handling techniques while learning without losing representational power. Similarly, in [2], a system is presented to learn the 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. ILP can learn well when the positive and negative examples of states before all target actions are given. However, in our problem, there are only example traces with partial (or empty) states provided as input. To the best of our knowledge, no work so far has been done to apply ILP to our problem.

2.4. Markov Logic Networks

2.4.1. Summary

We use Markov Logic Networks (MLNs) [35,9,23] to help learn the action models in our work. MLN is a powerful framework that combines probability and first-order logic. An MLN consists of a set of weighted formulas and provide a way to soften hard constraints in first-order logic. The main motivation behind MLNs to "soften" these hard constraints is that when a world violates a formula in a knowledge base, it is less probable, but not impossible. Thus, each formula should have an associated weight that reflects how strong a constraint is. Each weight can be learned from data using a variety of methods, including convex optimization of the likelihood or a related function, iterative scaling and margin maximization [27]. The network structure can also be learned from the input data, typically by performing a greedy search over conjunctions of variables [21,28,20]. Furthermore, different versions of MLNs have been proposed. For instance, Wang and Domingos [43] introduce hybrid MLNs, in which continuous properties and functions over them can appear as features; Singla and Domingos [41] extend Markov logic to infinite domains by casting it in the framework of Gibbs measures.

2.4.2. Weight learning in MLNs

Since one of the major steps in our LAMP algorithm involves learning weights for a set of candidate formulas using standard techniques in MLNs, we briefly review one of the commonly used weight learning algorithms for optimizing the pseudo log-likelihood in MLNs.

In this work, we use the idea of weight learning in MLNs as presented in [35]. A brief description of this idea is given as follows. A Markov Logic Network L is a set of pairs (F_i, w_i), where F_i is a formula in first-order logic and w_i is a real number. Together with a finite set of constants $C = \{c_1, c_2, \ldots, c_{|C|}\}$, it defines a Markov network $M_{L,C}$ as follows:

1. $M_{L,C}$ contains one binary node for each possible grounding of each predicate appearing in L. The value of the node is 1 if the grounded predicate is true, and 0 otherwise.
2. $M_{L,C}$ contains one feature for each possible grounding of each formula F_i in L. The value of this feature is 1 if the ground formula is true, and 0 otherwise. The weight of the feature is the w_i associated with F_i in L.

We assume that all formulas are in clausal form. Let X be a set of all propositions describing a world, $\mathcal{F}$ be a set of all clauses in an MLN, w_i be a weight associated with a clause $f_i \in \mathcal{F}$. We optimize the pseudo likelihood [3], which is defined as:

$$P^*_w(X = x) = \prod_{l=1}^{n} P_w\big(X_l = x_l | MB_x(X_l)\big)$$

where X_l is a ground atom in X, $MB_x(X_l)$ is the state of the Markov blanket of X_l and n is the number of ground atoms in X. In a Markov network, the Markov blanket of a node is defined as a set of its neighboring nodes. Similarly, the Markov blanket of a ground atom is a set of ground atoms that appear in some instantiation of a formula with it. $P_w(X_l = x_l | MB_x(X_l))$ can be computed by:

$$P_w\big(X_l = x_l | MB_x(X_l)\big) = \frac{C_{(X_l = x_l)}}{C_{(X_l=0)} + C_{(X_l=1)}}$$

$$C_{(X_l = x_l)} = \exp \sum_{f_i \in F_l} w_i f_i\big(X_l = x_l, MB_x(X_l)\big)$$

F_l is the set of ground formulas that X_l appears in, and $f_i(X_l = x_l, MB_x(X_l))$ is the value (0 or 1) of the feature corresponding to the ith ground formula when $X_l = x_l$ and its Markov blanket is in the state of $MB_x(X_l)$. By optimizing the pseudo log-likelihood using the limited-memory BFGS algorithm [26], we can learn the weights for MLNs.

MLNs have been applied to several real-world applications. For instance, Domingos [7] proposes to apply Markov logic to model real social networks, which evolve in time with multiple types of arcs and nodes and are affected by the actions of multiple players; Singla et al. [40] build an integrated solution based on Markov logic to solve the entity resolution problem, which is about how to determine which records in a database refer to the same entities; furthermore, Poon et al. [34] propose a joint approach to perform information extraction using Markov logic and existing algorithms, where segmentation of all records and entity resolution are performed together in a single integrated inference process; Domingos [8] gives a theoretical discussion on the relationship of data mining and Markov logic; Lowd et al. [22] present an unsupervised approach to extract semantic networks from large volumes of text by applying Markov logic, etc.

3. Problem definition

A classical planning problem is described as $\mathcal{P} = (\Sigma, s_0, g)$, where $\Sigma = (S, \mathcal{A}, \gamma)$ is a planning domain, S is a set of states, $\mathcal{A}$ is a set of action models, γ is a deterministic transition function: $S \times A \to S$, s_0 is an initial state, and g is a goal description. An action model is defined as an action heading with preconditions and effects, where an action heading is composed of an action name with zero or more parameters. A plan is an action sequence $(a_0, a_1, \ldots, a_n)$ which makes a projection from s_0 to g. A plan trace is defined as $T = (s_0, a_0, s_1, a_1, \ldots, s_n, a_n, g)$, where $s_1, \ldots, s_n$ are partial observations of intermediate states; that is, each s_i is a subset of a world state. These partial observations are allowed to be empty. Each a_i is an instantiated action heading in the form of ‘action name(parameters)’ such as ‘move(L1 home)’ (that is, move is the name of the action, L1 and home are parameters describing two *location*s).

Our learning problem can be stated as follows. We are given a set of plan traces $\mathcal{T}$, a set of predicates and a set of action headings A that occur in $\mathcal{T}$. As output, LAMP outputs preconditions and effects of each action heading in A. An example of the input and desired output of the algorithm LAMP from the *briefcase* domain is shown in Tables 1 and 2.[4]

4. The LAMP algorithm

In this section, we give detailed descriptions of our LAMP algorithm. We first give an overview of our algorithm, as shown in Algorithm 1.

Algorithm 1 Overview of LAMP.

```
Input: (1) A set of predicates $P$; (2) A set of action headings $A$; (3) A set of plan traces $T$.
Output: A set of action models $\mathcal{A}$.
1: Encode each plan trace as a set of propositions, denoted as DB (i.e., database), and all the plan traces are denoted as DBs: $DBs = encode\_traces(T)$;
2: Generate candidate formulas $F$ according to our correctness constraints F1 to F5: $F = generate\_formulas(P, A)$;
3: Learn weights $W$ of all the candidate formulas: $W = learn\_weights(F, DBs)$;
4: Convert the weighted candidate formulas to action models $\mathcal{A}$:
   $\mathcal{A} = attain\_models(W, F)$;
5: return $\mathcal{A}$;
```

In the following subsections, we will give a detailed description of each step that corresponds to the *italic* parts in Algorithm 1.

4.1. Step 1: encode plan traces

In the first step of Algorithm 1, the procedure *encode_traces* encodes all the plan traces as a set of proposition databases *DB*s with plan traces T as input. To encode plan traces as propositional formulas, states and state transitions need be encoded. A brief description of this procedure is given in Algorithm 2. In the following, we give the detailed description of two main steps (Steps 5 and 9) in Algorithm 2.

Encode a state as a propositional formula In Step 5 of Algorithm 2, our idea is to use a propositional formula to represent facts that hold in a state. For instance, consider the *briefcase* domain in Table 1 where we have an object *o1* and a location *l1*. The state, describing that the object *o1* is in a briefcase and the briefcase is at a location *l1*, can be represented with the formula: *in(o1)* $\wedge$ *is-at(l1)*. If we consider *in(o1)* and *is-at(l1)* as propositional variables, the formula is a propositional

[4] For clarity and for readers who are not familiar with this domain, here we briefly describe what the predicates and actions in the *briefcase* domain mean. In this domain, we have some portable objects and several locations, together with a briefcase which can be used for moving objects between one place and another. For predicates, *(at ?x - portable ?l - location)* means the portable object x is at a location l, not necessarily in the briefcase, *(in ?x - portable)* means the portable object x is currently in the briefcase and *(is-at ?l - location)* means the briefcase is at location l. For actions, *(move ?m - location ?l - location)* means a person had moved the briefcase from location m to location l, *(take-out ?x - portable)* means the portable object x is taken out of the briefcase and *(put-in ?x - portable ?l - location)* means the portable object x is put into the briefcase and the briefcase is at location l.

Table 1
The input of LAMP.

Input: predicates
(at ?x - portable ?l - location) (in ?x - portable) (is-at ?l - location)

Input: action headings
(move ?m - location ?l - location) (take-out ?x - portable) (put-in ?x - portable ?l - location)

Input: plan traces

	Plan trace 1	Plan trace 2	Plan trace 3
initial state:	(is-at l1) (at o1 l1) (at o2 l2) (not (is-at l2)) (not (at o1 l2)) (not (at o2 l1)) (not (in o1)) (not (in o2))	(is-at home) (at o1 l1) (not (is-at l1)) (not (at o1 home)) (not (in o1))	(is-at l2) (at o1 l2) (at o2 home) (not (is-at l1)) (not (at o1 home)) (not (at o2 l2)) (not (in o1)) (not (in o2))
action 1:	(put-in o1 l1)	(move home l1)	(put-in o1 l2)
observation 1:	(is-at l1)		
action 2:	(move l1 l2)	(put-in o1 l1)	(move l2 l1)
observation 2:		(in o1)	
action 3:	(put-in o2 l2)	(move l1 home)	(take-out o1)
observation 3:	(in o2) (is-at l2)		
action 4:	(move l2 home)		(move l1 home)
observation 4:			
action 5:			(put-in o2 home)
observation 5:			(in o2)
action 6:			(move home l2)
goal state:	(is-at home) (at o1 home) (at o2 home)	(is-at home) (at o1 home)	(is-at l1) (at o1 l1) (at o2 l2)

Table 2
The desired output of LAMP.

(move ?m - location ?l - location)	
preconditions	(is-at ?m)
effects	(and (is-at ?l) (not (is-at ?m)) (forall (?x - portable) (when (in ?x) (and (at ?x ?l)(not (at ?x ?m))))))
(put-in ?x - portable ?l - location)	
preconditions	(and (at ?x ?l) (is-at ?l))
effects	(in ?x)
(take-out ?x - portable)	
preconditions	(in ?x)
effects	(not (in ?x))

formula. A model of the propositional formula is the one that assigns true to the propositional variables *in(o1)* and *is-at(l1)*. If we have one more location *l2*, the above propositional formula should be modified as: *in(o1)*$\wedge$ *is-at(l1)* $\wedge$ $\neg$*is-at(l2)*. Notice that we use $\neg p$ when referring to a logic formula, while (not p) when referring to a PDDL description; e.g., $\neg$*is-at(l2)* will be described as *(not (is-at l2))* in PDDL.

Encode an action as a proposition In Step 9 of Algorithm 2, our idea is to encode an action (state transitions) in a plan trace into a proposition, in a way similar to situation calculus [25]. The behavior of deterministic actions is described by the transition function $\gamma : S \times A \rightarrow S$. For instance, the action *move(l1,l2)* in Table 2 is described by $\gamma(s_1, move(l1, l2)) = s_2$.

Algorithm 2 Encode plan traces: *encode_traces(T)*.

```
Input: A set of plan traces $T$.
Output: A set of databases $DBs$.
 1: Initialize $DBs = \emptyset$;
 2: for each plan trace $pt \in T$ do
 3:     Initialize $DB = \emptyset$;
 4:     for each state $s \in pt$ do
 5:         Encode $s$ as a formula, which is a conjunction of propositions;
 6:         Put all the propositions of the conjunction in $DB$;
 7:     end for
 8:     for each action $a \in t$ do
 9:         Encode $a$ as a proposition;
10:         Put the proposition in $DB$;
11:     end for
12:     Put $DB$ in $DBs$;
13: end for
14: return $DBs$;
```

In s_1, the briefcase is at location *l1*, while in s_2, it is at *l2*. The states s_1 and s_2 can be represented as: *is-at(l1)* $\wedge$ $\neg$*is-at(l2)* and $\neg$*is-at(l1)* $\wedge$ *is-at(l2)*. These formulas, however, cannot be used to represent the fact that the system *evolves* from a state s_1 to a state s_2. We need a propositional formula to assert that, in state s_1, *is-at(l1)* $\wedge$ $\neg$*is-at(l2)* holds, and in state s_2, after executing the action, $\neg$*is-at(l1)* $\wedge$ *is-at(l2)* holds. We need different propositional variables that hold in different states to specify that a fact holds in one state but does not hold in the other state.

In Step 9 of Algorithm 2, we generate situation-labels s_i for ground atoms in the following way. We denote each trace as (s_0, a_0, ..., a_{n-1}, s_n), and assign a situation-label s_i to a ground atom p if p appears in the state s_i. Thus, if a ground atom *is-at(l1)* appears before an action a_1, it is then associated with the situation-label s_1, represented as one of the atom's parameters, i.e., *is-at(l1,* s_1*)*. The parameters of each candidate formula are determined by the parameters of actions or predicates; i.e., they are composed of the original parameters of actions or predicates plus the situation-label variable i to denote s_i.

By introducing new state parameters s_i, we can get two different literals: *is-at(l1,* s_1*)* and *is-at(l2,* s_2*)*, before and after an action *move(l1, l2,* s_1*)*, respectively. The action states that a briefcase is at a location *l1* in state s_1, and it is at location *l2* in another state s_2 after the action *move(l1, l2,* s_1*)*. Thus, the fact that the system evolves from the state s_1 to the state s_2 can be represented by the statement: *is-at(l1,* s_1*)* $\wedge$ $\neg$*is-at(l2,* s_1*)* $\wedge$ $\neg$*is-at(l1,* s_2*)* $\wedge$ *is-at(l2,* s_2*)*. This formula encodes the transition from state s_1 to s_2.

Furthermore, we represent the fact that it is the action *move(l1, l2)* that causes the transition by a propositional variable *move(l1, l2,* s_1*)*, which holds true if the action *move* is executed in state s_1. As a result, we can now represent the function $\gamma(s_1, move(l1, l2))$ as

$$move(l1, l2, s_1) \wedge \text{is-at}(l1, s_1) \wedge \neg\text{is-at}(l2, s_1) \wedge \neg\text{is-at}(l1, s_2) \wedge \text{is-at}(l2, s_2)$$

Thus, we can naturally encode plan traces as propositional formulas. For instance, we can encode the second plan trace of Table 1 with the following formula:

$$\begin{aligned} &\textit{is-at}(home, s_0) \wedge at(o1, l1, s_0) \wedge \neg \textit{is-at}(l1, s_0) \wedge \neg at(o1, home, s_0) \wedge \neg in(o1, s_0) \\ &\quad \wedge \text{move}(home, l1, s_0) \wedge \text{put-in}(o1, l1, s_1) \wedge in(o1, s_2) \wedge \text{move}(l1, home, s_2) \\ &\quad \wedge \textit{is-at}(home, s_3) \wedge at(o1, home, s_3) \end{aligned}$$

Since each plan trace can be encoded as a conjunction of grounded literals (also called *facts*), it is equivalent to encoding a plan trace as a database (referred to as a *DB* below). Each record in a *DB* is a fact. Records are related in a conjunction. Thus, we can encode different plan traces as different *DB*s. As an example, the plan traces in Table 1 can be encoded into *DB*s as shown in Table 3, where the state symbol s_i is represented with an integer i. Notice that we do not use closed-world assumption. The *truth* value of a proposition not recorded in Table 3 is left as *unknown*. We will learn the weights of formulas by counting the number of formulas that are satisfied by the *known* grounded literals recorded in *DB*s. We will give the detailed description in Section 4.3.

Notice that in the above formulation, we are not imposing any constraints on how much the intermediate state information needs be observed. Since the proportion of intermediate state information, in our representation, is encoded into a number of facts in *DB*s, less intermediate state information will only affect the number of records in *DB*s. However, as we show later in our experiments, the proportion of intermediate state information that we have in the input plan traces will affect how accurately we can learn the action models; as we will show later, in general, the more intermediate state information we have, the better the learning accuracy will be.

Table 3
Encodings of plan traces into *DB*s (notice that a proposition preceded by the notation "!" means the proposition is *false*, a proposition existing in *DB* means it is *true*, and a proposition not existing in *DB* means it is *unknown*), which is consistent with the representation of MLNs' input.

DB 1	*DB* 2	*DB* 3
(is-at l1 0)	(is-at home 0)	(is-at l2 0)
(at o1 l1 0)	(at o1 l1 0)	(at o1 l2 0)
(at o2 l2 0)	!(is-at l1 0)	(at o2 home 0)
!(is-at l2 0)	!(at o1 home 0)	!(is-at l1 0)
!(at o1 l2 0)	!(in o1 0)	!(at o1 home 0)
!(at o2 l1 0)	(move home l1 0)	!(at o2 l2 0)
!(in o1 0)	(put-in o1 l1 1)	!(in o1 0)
!(in o2 0)	(in o1 2)	!(in o2 0)
(put-in o1 l1 0)	(move l1 home 2)	(put-in o1 l2 0)
(is-at l1 1)	(is-at home 3)	(move l2 l1 1)
(move l1 l2 1)	(at o1 home 3)	(take-out o1 2)
(put-in o2 l2 2)		(move l1 home 3)
(in o2 3)		(put-in o2 home 4)
(is-at l2 3)		(in o2 5)
(move l2 home 3)		(move home l2 5)
(is-at home 4)		(is-at l1 6)
(at o1 home 4)		(at o1 l1 6)
(at o2 home 4)		(at o2 l2 6)

4.2. Step 2: generating candidate formulas

In the previous step, we encoded each plan trace into a propositional formula, which is a conjunction of ground literals and then represented the formula with a database whose records are facts. In this step, we will describe how to generate the candidate formulas, which is done by the procedure *generate_formulas* with a set of predicates P and a set of action headings A as input.

There are five parts of an action model to be learned: preconditions, positive effects, negative effects, positive conditional effects and negative conditional effects. Our idea is to describe each part with a set of formulas. Suppose that we describe the possible preconditions of an action with a set of formulas F. If a formula $f \in F$, in the form of "$a \to p$" holds, then the action a has a precondition p.

As an overview, our basic algorithm for generating candidate formulas can be described as follows (also see Algorithm 3).

- Impose specific correctness constraints, written in the form of "$a \to b$", where a is an action, and b is a logical formula. Encode some specific requirements of preconditions, positive and negative effects as well as positive and negative *conditional* effects in the form of implications to make sure the formulas we generate can reflect the correctness requirements.
- Enumerate all possibilities to ground the candidate formulas in the previous step. For example, we will replace a propositional variable with a predicate in the predicate list and enumerate all possibilities of the possible grounding. This allows us to anticipate all forms of preconditions in an action model.
- Enumerate the cases when the parameters in all candidate formulas might be quantified.

From these steps, we can generate a number of candidate formulas with quantifiers as well as logical implications. We then employ MLN-learning algorithms to select a subset of these formulas and convert it to action models. These ideas are described in detail in the following (see formulas **F1** to **F5** below).

We conform to the requirements in PDDL in that a precondition can be a literal or an implication with or without a universal quantifier. Preconditions are conjunctive. An action's effect can be a positive or a negative literal with or without universal quantifiers. An effect can also be a conditional effect with or without an existential quantifier to quantify its condition part. The relationship between effects is also conjunctive.

Algorithm 3 Generate candidate formulas: *generate_formulas(P, A)*.

```
Input: (1) A set of predicates $P$, (2) a set of actions $A$.
Output: A set of candidate formulas $F$.
1: Initialize $F = \emptyset$;
2: for each action $a \in A$ do
3:     Generate candidate formulas according to F1–F5 with $a$ and $P$;
4:     Put the generated candidate formulas to $F$;
5: end for
6: return $F$;
```

We now give a detailed description of Step 3 in Algorithm 3, which is divided into five parts in formulas **F1** to **F5**.

Table 4
Candidate formulas according to formula (1a): encoding preconditions.

ID	Formulas
1	$\forall i.p.$((take-out o i) $\rightarrow$ (at o p i))
2	$\forall i.x.p.$((take-out o i) $\rightarrow$ (at x p i))
3	$\forall i.p.$((take-out o i) $\rightarrow$ (is-at p i))
4	$\forall i.$((take-out o i) $\rightarrow$ (in o i))
5	$\forall i.x.$((take-out o i) $\rightarrow$ (in x i))

[F1] *Preconditions.* Any literal p can be a precondition of an action a at a state i before that action in a plan trace. According to this formula-generation method, we build candidate formulas in the form of formulas (1a) and (1b).

- For each action a and literal p, and each state i just before a, we generate the following formula:

  $$\forall i.\xi.\big(a(i) \rightarrow p(\xi, i)\big) \tag{1a}$$

  where $\xi = \{x_1, x_2, \ldots, x_n\}$ is a set of parameters that do not appear in a's parameters but exist in p's parameters ($n \geqslant 0$) (i.e., ξ represents the free variables in a precondition of an action given the action heading), i is the ith state s_i (likewise for formulas (1b)–(5)). For simplicity, we omit a's parameters and p's parameters shared by a in this representation.
  Corresponding to each formula above, we also generate candidate formulas with *universal quantifiers* for the preconditions of the action a. Let c be a set of parameters common to both a and p and $\xi = \{x_1, x_2, \ldots, x_n\}$ be parameters of p that do not appear in a's parameter list. We can generate "forall ?ξ (p ?ξ ?c)". When a includes all the parameters of p, we generate "(p ?c)" as a candidate for the precondition.
- We generate a precondition of an action a as an implication, which is denoted as $p_1 \rightarrow p_2$ (in PDDL, it is equally denoted as "(imply p_1 p_2)"). We generate the following formula:

  $$\forall i.\xi.\big(a(i) \rightarrow \big(p_1(\xi_1, i) \rightarrow p_2(\xi_2, i)\big)\big)$$

  or equivalently,

  $$\forall i.\xi.\big(\big(a(i) \wedge p_1(\xi_1, i)\big) \rightarrow p_2(\xi_2, i)\big) \tag{1b}$$

  where $\xi = \xi_1 \cup \xi_2$. Likewise, for an implication in the form of "$p_1 \rightarrow \neg p_2$" or "$\neg p_1 \rightarrow p_2$", we will similarly generate its corresponding formulas accordingly. Corresponding to formula (1b), we can build a precondition with an implication as "forall ?ξ (imply (p_1 ?ξ_1 ?c_1) (p_2 ?ξ_2 ?c_2))", where c_1 and c_2 are two sets of parameters appearing in a's. We only consider implications in the form of "$p_1 \rightarrow p_2$", "$\neg p_1 \rightarrow p_2$" or "$p_1 \rightarrow \neg p_2$", although it is easy to extend an implication to a more complex form such as $p_1^{(1)} \wedge p_1^{(2)} \wedge \cdots \wedge p_1^{(t)} \rightarrow p_2$ (in PDDL, it is denoted as "(imply (and $p_1^{(1)}$ $p_1^{(2)}$ … $p_1^{(t)}$) p_2)").

These formulas are called *candidate formula*s because only some of them will be chosen in the end as preconditions of corresponding actions. We will select a subset of these candidate formulas and transform them to action models in Steps 3 and 4. In the following example, we show how to generate candidate formulas according to formulas (1a) and (1b).

Example 1. We generate candidate formulas for the action "take-out" in Table 1 according to formulas (1a) and (1b). The result is shown in Tables 4 and 5. The initial weights of the generated candidate formulas are zeros. Candidate formulas for other actions in Table 1 can be generated similarly.

[F2] *Positive effects.* We generate positive effect formulas as follows. For each literal p, p is a positive effect of an action a (that is, a literal without a negation symbol) and p is added to the state right after a in a plan trace. For instance, if we generate a candidate formula for "(is-at ?l)" to be a positive effect of "(move ?m ?l)" in Table 2, then "(is-at l2)" is added after "(move l1 l2)" is executed in plan trace 1 of Table 1.

To avoid generating lengthy effect lists for action models, we also generate a preference constraint that p does not already hold when p is added, since otherwise, many effects would be redundantly generated. Of course, there are cases when this constraint false, and thus we only state this constraint as a preferred one; the preferences are reflected by the weights attached to these formulas which are to be learned. This is the motivation for us to generate candidate formulas in the form of formula (2), which means that $p(\xi)$ does not hold in state s_i and will hold in state s_{i+1}.

$$\forall i.\xi.\big(a(i) \rightarrow \neg p(\xi, i) \wedge p(\xi, i+1)\big) \tag{2}$$

Corresponding to this formula, we can generate candidate formulas with *universal quantifiers*. Recall that both positive and negative effects, if not conditional effects, do not support existential quantifiers as in PDDL definition. We build positive effects as follows: "forall ?ξ (p ?ξ ?c)". When the action a includes all the parameters of the predicate p, we can generate

Table 5
Candidate formulas according to formula (1b): encoding preconditions.

ID	Formulas
1	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \to (\text{at } x\ p\ i))$
2	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \to (\text{is-at } p\ i))$
3	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \to (\text{in } o\ i))$
4	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \to (\text{in } x\ i))$
5	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \to (\text{at } o\ p\ i))$
6	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \to (\text{is-at } p\ i))$
7	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \to (\text{in } o\ i))$
8	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \to (\text{in } x\ i))$
9	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \to (\text{at } o\ p\ i))$
10	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \to (\text{at } x\ p\ i))$
11	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \to (\text{in } o\ i))$
12	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \to (\text{in } x\ i))$
13	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \to (\text{at } o\ p\ i))$
14	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \to (\text{at } x\ p\ i))$
15	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \to (\text{is-at } p\ i))$
16	$\forall i.x.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \to (\text{in } x\ i))$
17	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \to (\text{at } o\ p\ i))$
18	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \to (\text{at } x\ p\ i))$
19	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \to (\text{is-at } p\ i))$
20	$\forall i.x.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \to (\text{in } o\ i))$

Table 6
Candidate formulas according to formula (2): encoding positive effects.

ID	Formulas
1	$\forall i.p.((\text{take-out } o\ i) \to (\text{at } o\ p\ i+1) \wedge \neg(\text{at } o\ p\ i))$
2	$\forall i.x.p.((\text{take-out } o\ i) \to (\text{at } x\ p\ i+1) \wedge \neg(\text{at } x\ p\ i))$
3	$\forall i.p.((\text{take-out } o\ i) \to (\text{is-at } p\ i+1) \wedge \neg(\text{is-at } p\ i))$
4	$\forall i.((\text{take-out } o\ i) \to (\text{in } o\ i+1) \wedge \neg(\text{in } o\ i))$
5	$\forall i.x.((\text{take-out } o\ i) \to (\text{in } x\ i+1) \wedge \neg(\text{in } x\ i))$

Table 7
Candidate formulas according to formula (3): encoding negative effects.

ID	Formulas
1	$\forall i.p.((\text{take-out } o\ i) \to \neg(\text{at } o\ p\ i+1) \wedge (\text{at } o\ p\ i))$
2	$\forall i.x.p.((\text{take-out } o\ i) \to \neg(\text{at } x\ p\ i+1) \wedge (\text{at } x\ p\ i))$
3	$\forall i.p.((\text{take-out } o\ i) \to \neg(\text{is-at } p\ i+1) \wedge (\text{is-at } p\ i))$
4	$\forall i.((\text{take-out } o\ i) \to \neg(\text{in } o\ i+1) \wedge (\text{in } o\ i))$
5	$\forall i.x.((\text{take-out } o\ i) \to \neg(\text{in } x\ i+1) \wedge (\text{in } x\ i))$

positive effects in the PDDL form: "(p ?c)". Example 2 below illustrates how to generate candidate formulas according to formula (2):

Example 2. For different literals p and actions a, we can generate different formulas according to formula (2). The initial weights of the generated candidate formulas are zeros. In Table 6, we list all candidate formulas generated for the action "take-out".

[F3] *Negative effects.* For each action a, we generate p as a negative effect of an action a, and the condition that p will not be satisfied (will be deleted) in the state after a is executed. As in the case of F2, to limit the action model size, we also have a preference constraint that p can be deleted only when p exists. For instance, "(is-at ?m)" is a negative effect of "(move ?m ?l)", then "(is-at l1)" is satisfied in the state where "(move l1 l2)" is executed and is not satisfied in the state after that. Thus, we build the corresponding formulas in the form of formula (3).

$$\forall i.\xi.\big(a(i) \to p(\xi, i) \wedge \neg p(\xi, i+1)\big) \tag{3}$$

Corresponding to this formula, we can generate candidate formulas with *universal quantifiers* and build negative effects in the PDDL form of "forall ?ξ (not (p ?ξ ?c))". When a includes all the parameters of p, we build negative effects in the PDDL form of "(not (p ?c))". In the following, Example 3 will show how to generate candidate formulas according to formula (3).

Example 3. According to formula (3), we generate candidate formulas for the action "take-out" in Table 1. The initial weights of the generated candidate formulas are zeros. The result is shown in Table 7.

Table 8
Candidate formulas according to formula (4): encoding positive conditional effects.

ID	Formulas
1	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \rightarrow \neg(\text{at } x\ p\ i) \wedge (\text{at } x\ p\ i+1))$
2	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \rightarrow \neg(\text{is-at } p\ i) \wedge (\text{is-at } p\ i+1))$
3	$\forall i.\exists p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \rightarrow \neg(\text{in } o\ i) \wedge (\text{in } o\ i+1))$
4	$\forall i.x.\exists p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \rightarrow \neg(\text{in } x\ i) \wedge (\text{in } x\ i+1))$
5	$\forall i.p.\exists x.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \rightarrow \neg(\text{at } o\ p\ i) \wedge (\text{at } o\ p\ i+1))$
6	$\forall i.p.\exists x.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \rightarrow \neg(\text{is-at } p\ i) \wedge (\text{is-at } p\ i+1))$
7	$\forall i.\exists x.p.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \rightarrow \neg(\text{in } o\ i) \wedge (\text{in } o\ i+1))$
8	$\forall i.x.\exists p.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \rightarrow \neg(\text{in } x\ i) \wedge (\text{in } x\ i+1))$
9	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \rightarrow \neg(\text{at } o\ p\ i) \wedge (\text{at } o\ p\ i+1))$
10	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \rightarrow \neg(\text{at } x\ p\ i) \wedge (\text{at } x\ p\ i+1))$
11	$\forall i.\exists p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \rightarrow \neg(\text{in } o\ i) \wedge (\text{in } o\ i+1))$
12	$\forall i.x.\exists p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \rightarrow \neg(\text{in } x\ i) \wedge (\text{in } x\ i+1))$
13	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \rightarrow \neg(\text{at } o\ p\ i) \wedge (\text{at } o\ p\ i{+}1))$
14	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \rightarrow \neg(\text{at } x\ p\ i) \wedge (\text{at } x\ p\ i+1))$
15	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \rightarrow \neg(\text{is-at } p\ i) \wedge (\text{is-at } p\ i{+}1))$
16	$\forall i.x.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \rightarrow \neg(\text{in } x\ i) \wedge (\text{in } x\ i+1))$
17	$\forall i.p.\exists x.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \rightarrow \neg(\text{at } o\ p\ i) \wedge (\text{at } o\ p\ i+1))$
18	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \rightarrow \neg(\text{at } x\ p\ i) \wedge (\text{at } x\ p\ i+1))$
19	$\forall i.p.\exists x.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \rightarrow \neg(\text{is-at } p\ i) \wedge (\text{is-at } p\ i+1))$
20	$\forall i.\exists x.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \rightarrow \neg(\text{in } o\ i) \wedge (\text{in } o\ i{+}1))$

[F4] *Positive conditional effects.* For each action a we generate positive conditional effects of a as "(when p_1 p_2)", meaning if p_1 holds, p_2 will also hold after a is executed. We can assert that, if "(when p_1 p_2)" is a positive conditional effect of an action a, then p_2 holds in the state after a's execution if p_1 holds in the state before a's execution. We also have a preference constraint that as in **F2**, p_2 is preferred not to hold before a's execution. Thus, we build the following candidate formulas.

$$\forall i.\xi.\xi_2.\exists \xi_1.\big(a(i) \wedge p_1(\xi, \xi_1, i) \rightarrow \neg p_2(\xi, \xi_2, i) \wedge p_2(\xi, \xi_2, i+1)\big) \tag{4}$$

where ξ, ξ_1 and ξ_2 are sets of parameters not appearing in a's parameters, and $\xi \cap \xi_1 = \emptyset \wedge \xi \cap \xi_2 = \emptyset \wedge \xi_1 \cap \xi_2 = \emptyset$. We generate a positive conditional effect in this form as shown in formula (4), although it is easy to extend it to other forms such as "$\forall i.\xi.\xi_1.\xi_2.(a(i) \wedge p_1(\xi, \xi_1, i) \rightarrow \neg p_2(\xi, \xi_2, i) \wedge p_2(\xi, \xi_2, i+1))$", or more complex forms corresponding to other positive conditional effects such as "(when $(p_1^1 \wedge p_1^2 \wedge \ldots)$ p_2)".

Corresponding to formula (4), we can generate candidate formulas with both *conditional effects*, *universal quantifiers* as well as *existential quantifiers* and build positive conditional effects in PDDL, "forall ?ξ ?ξ_2 (when (exist ?ξ_1 (p_1 ?ξ ?ξ_1 ?c_1))(p_2 ?ξ ?ξ_2 ?c_2))", or "(when (p_1 ?c_1)(p_2 ?c_2))" when ξ, ξ_1 and ξ_2 are all empty sets (notice that each one of them can be either empty or not, which will correspond to different PDDL forms), where c_1 and c_2 are two sets of parameters appearing in a's. In the following, Example 4 will show how to generate candidate formulas based on formula (4).

Example 4. For the action "take-out" in Table 1, we generate candidate formulas according to formula (4). The initial weights of the generated candidate formulas are zeros. The result is shown in Table 8.

[F5] *Negative conditional effects.* Similar to **F4**, for each action a, we generate negative conditional effects of a of the form "(when p_1 $\neg p_2$)", meaning that if p_1 holds before a's execution, then p_2 will never hold (will be deleted) after a is executed. Additionally, we add a preference that if p_2 is deleted in an action's effect then p_2 exists in the precondition of the action. We formulate this as follows.

$$\forall i.\xi.\xi_2.\exists \xi_1.\big(a(i) \wedge p_1(\xi, \xi_1, i) \rightarrow p_2(\xi, \xi_2, i) \wedge \neg p_2(\xi, \xi_2, i+1)\big) \tag{5}$$

We generate negative conditional effects of this form as shown in formula (5), similar to **F4**. Corresponding to formula (5), we can generate candidate formulas with *conditional effects, universal and existential quantifiers* and build negative conditional effects in PDDL, "forall ?ξ ?ξ_2 (when (exist ?ξ_1 (p_1 ?ξ ?ξ_1 ?c_1))(not (p_2 ?ξ ?ξ_2 ?c_2)))", or "(when (p_1 ?c_1)(not (p_2 ?c_2)))" when ξ, ξ_1 and ξ_2 are all empty sets, or other corresponding forms when some of them being empty. Likewise, Example 5 will show how to generate candidate formulas based on formula (5).

Example 5. For the action "take-out" in Table 1, we generate candidate formulas according to formula (5). The initial weights of the generated candidate formulas are zeros. The result is shown in Table 9.

The above formulas are responsible for generating the space of all possible candidate literals and formulas for action models, in terms of their preconditions and effects, but constraints are then needed for a learning system to select a set of consistent formulas that are consistent with the training examples, which are the plan traces. We impose the following three constraints below.

Table 9
Candidate formulas according to formula (5): encoding negative conditional effects.

ID	Formulas
1	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \rightarrow (\text{at } x\ p\ i) \wedge \neg(\text{at } x\ p\ i+1))$
2	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \rightarrow (\text{is-at } p\ i) \wedge \neg(\text{is-at } p\ i+1))$
3	$\forall i.\exists p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \rightarrow (\text{in } o\ i) \wedge \neg(\text{in } o\ i+1))$
4	$\forall i.x.\exists p.((\text{take-out } o\ i) \wedge (\text{at } o\ p\ i) \rightarrow (\text{in } x\ i) \wedge \neg(\text{in } x\ i+1))$
5	$\forall i.p.\exists x.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \rightarrow (\text{at } o\ p\ i) \wedge \neg(\text{at } o\ p\ i+1))$
6	$\forall i.p.\exists x.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \rightarrow (\text{is-at } p\ i) \wedge \neg(\text{is-at } p\ i+1))$
7	$\forall i.\exists x.p.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \rightarrow (\text{in } o\ i) \wedge \neg(\text{in } o\ i+1))$
8	$\forall i.x.\exists p.((\text{take-out } o\ i) \wedge (\text{at } x\ p\ i) \rightarrow (\text{in } x\ i) \wedge \neg(\text{in } x\ i+1))$
9	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \rightarrow (\text{at } o\ p\ i) \wedge \neg(\text{at } o\ p\ i+1))$
10	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \rightarrow (\text{at } x\ p\ i) \wedge \neg(\text{at } x\ p\ i+1))$
11	$\forall i.\exists p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \rightarrow (\text{in } o\ i) \wedge \neg(\text{in } o\ i+1))$
12	$\forall i.x.\exists p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \rightarrow (\text{in } x\ i) \wedge \neg(\text{in } x\ i+1))$
13	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \rightarrow (\text{at } o\ p\ i) \wedge \neg(\text{at } o\ p\ i+1))$
14	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \rightarrow (\text{at } x\ p\ i) \wedge \neg(\text{at } x\ p\ i+1))$
15	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \rightarrow (\text{is-at } p\ i) \wedge \neg(\text{is-at } p\ i+1))$
16	$\forall i.x.((\text{take-out } o\ i) \wedge (\text{in } o\ i) \rightarrow (\text{in } x\ i) \wedge \neg(\text{in } x\ i+1))$
17	$\forall i.p.\exists x.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \rightarrow (\text{at } o\ p\ i) \wedge \neg(\text{at } o\ p\ i+1))$
18	$\forall i.x.p.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \rightarrow (\text{at } x\ p\ i) \wedge \neg(\text{at } x\ p\ i+1))$
19	$\forall i.p.\exists x.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \rightarrow (\text{is-at } p\ i) \wedge \neg(\text{is-at } p\ i+1))$
20	$\forall i.\exists x.((\text{take-out } o\ i) \wedge (\text{in } x\ i) \rightarrow (\text{in } o\ i) \wedge \neg(\text{in } o\ i+1))$

[A1] *Action-consistency constraint.* First, we wish the model learned do not conflict with themselves. Thus, if an action a has an effect p, then this same action a cannot have an effect $\neg p$ in the same state after a is executed. We formulate this constraint as follows. For each action a and literal p, if in its effect there are two formulas "$f_1 = L1 \rightarrow p$" and"$f_2 = L2 \rightarrow \neg p$", where $L1$ and $L2$ are both a conjunction of literals, then we require that $L1$ and $L2$ are mutually exclusive. Notice that either $L1$ or $L2$ can be empty, i.e., "$f_1 = p$" or "$f_2 = \neg p$". $L1$ and $L2$ are not mutually exclusive when either of them is empty. For example, "(in ?x)" and "(not (in ?x))" should not be chosen as the effects of the action "(take-out ?x)" at the same time.

[A2] *Plan-consistency constraint.* We require that the action models learned are consistent with the training plan traces. This constraint is imposed on the relationship between ordered actions in plan traces, and it ensures that the causal links in the plan traces are not broken. That is, for each precondition p of an action a_j in a plan trace, either p is in the initial state, or there is an action a_i $(i < j)$ prior to a_j that adds p and there is no action a_k $(i < k < j)$ between a_i and a_j that deletes p. For each literal q in a state s_j, either q is in the initial state s_0, or there is an action a_i before s_j that adds q while no action a_k deletes q. We formulate the constraint as follows.

$$p \in \text{PRE}(a_j) \wedge p \in \text{EFF}(a_i) \wedge \neg p \notin \text{EFF}(a_k) \tag{6a}$$

and

$$q \in s_j \wedge \big(q \in s_0 \vee \big(q \in \text{EFF}(a_i) \wedge \neg q \notin \text{EFF}(a_k)\big)\big) \tag{6b}$$

where $i < k < j$, $\text{PRE}(a_j)$ is a set of preconditions of the action a_j and the state s_j is composed of a set of propositions (or predicates).

For example, for the action "put-in", since "is-at" is a precondition of "put-in", if it is not in the initial state, there should be an action "move" that adds it, and it is never deleted by the actions (if any) between "move" and "put-in". Otherwise, the action "put-in" cannot be executed after the action "move".

[A3] *Non-empty constraint.* We wish to avoid the extreme situation where in a plan trace, the action models are learned such that all except one of the actions in the trace have non-empty preconditions and effects. Although such an action model is not incorrect, it is rather undesirable. To avoid such cases, we require that the preconditions and effects of the actions we learn should be non-empty. In other words, for each action model a, the following formula should hold:

$$\text{PRE}(a) \neq \emptyset \wedge \text{EFF}(a) \neq \emptyset \tag{7}$$

These constraints can be used in the learning phase for building the action models in an MLN, or it can be used as post-processing constraints for selecting the effects after they are learned in Step 3 (see next subsection). In our experiment, these constraints are enforced.

4.3. Step 3: learning the weights of candidate formulas

In this section, we describe how to construct Markov Logic Networks [35,23] to learn the weights of the candidate formulas F based on the constraints A. The formulas with weights larger than a certain threshold will be chosen to represent preconditions and effects of the learned action models. This step will be done by the procedure *learn_weights* with candidate formulas F and databases *DB*s as input.

At a first glance, it might seem that using a maximum satisfiability-based algorithm would solve the problem of selecting a good subset of formulas. However, this is not the case. The reason is that a weighted satisfiability-based algorithm will assign non-uniform weights to a collection of formulas that correspond to instantiations of these formulas, thus they cannot be grouped into formulas expressing quantifiers or implications, because each of these complex formulas corresponds to a collection of ground instantiations. The weights on each instantiation will be different, making it impossible to combine them together to re-construct the first-order formulas. It is more natural to consider the weights of a first-order logic formula in its entirety. Therefore, a Markov Logic Network is a more appropriate model for us to learn the weights of the first-order logic formulas in a way to soften these hard constraints.

To learn the weights of formulas F, we exploit the *Alchemy* System of MLN [35,23] to calculate and optimize the score of `WPLL` (i.e., Weighted Pseudo Log-Likelihood) [3]. With respect to the weights w and a database x in *DB*s (a list of possible worlds), `WPLL` is defined as follows.

$$\begin{aligned}\mathrm{WPLL}(w,x) &= \log\prod_{l=1}^{n} P_w\big(X_l = x_l | MB_x(X_l)\big)\\ &= \sum_{l=1}^{n}\log P_w\big(X_l = x_l | MB_x(X_l)\big)\end{aligned}$$

where

$$P_w\big(X_l = x_l | MB_x(X_l)\big) = \frac{C_{(X_l=x_l)}}{C_{(X_l=0)} + C_{(X_l=1)}}$$

and $C_{(X_l=x_l)} = \exp\sum_{f_i\in F_l} w_i f_i(X_l = x_l, MB_x(X_l))$. n is the number of all the possible groundings of atoms appearing in all the formulas F, and X_l is the lth ground atom. $MB_x(X_l)$ is the state of the Markov blanket of X_l in x, where $x = (x_i)$ is a world state and x_i can be 1 or 0 which denotes the *truth* value of the corresponding ground atom (*true* or *false*, respectively). A Markov blanket of a ground atom is a set of ground atoms that appear in some grounding of a formula with it. F_l is the set of ground formulas that X_l appears in, and $f_i(X_l = x_l, MB_x(X_l))$ is the value (0 or 1) of the feature corresponding to the ith ground formula when $X_l = x_l$ and Markov blanket state $MB_x(X_l)$.

For instance, there is only one formula "$p(x, y) \to q(x)$" in F, and $x \in \{A, B\}$, $y \in \{C, D\}$. Then all the possible groundings are $\{p(A,C), p(A,D), p(B,C), p(B,D), q(A), q(B)\}$, i.e., $n = 6$ and X_l ($0 < l \leqslant n$) could be viewed as one of the groundings. $F_{p(A,C)}$ (or F_1) is $\{p(A,C) \to q(A)\}$, likewise for other F_l. A Markov blanket of $p(A,C)$ is $\{q(A)\}$, which can be easily found according to $F_{p(A,C)}$, likewise for Markov blankets of other ground atoms. Given a world state $x = (1,1,1,0,1,0)$, the value of $f_i(X_1 = x_1, MB_x(X_1))$ is 1 (note that X_1 is $p(A,C)$, x_1 is 1 in x, and $MB_x(X_1) = \{q(A)\}$ is also 1 in x). As a result,

$$C_{(X_1=x_1)} = \exp\sum_{f_i\in F_1} w_i f_i\big(X_1 = x_1, MB_x(X_1)\big) = e^{w_i}$$

Similarly, we can also calculate $C_{(X_1=0)} = e^{w_i}$. Furthermore, we have

$$\begin{aligned}\mathrm{WPLL}(w,x) &= \sum_{l=1}^{6}\log P_w\big(X_l = x_l | MB_x(X_l)\big)\\ &= \log\frac{e^{w_i}}{e^{w_i}+e^{w_i}} + \cdots\end{aligned}$$

where $w = w_i$ since there is only one formula, and the term $\log\frac{e^{w_i}}{e^{w_i}+e^{w_i}}$ is calculated when $l = 1$. Likewise for other l ($1 < l \leqslant 6$), we can calculate their corresponding values.

The score `WPLL` is used to measure the likelihood of all weighted candidate formulas to be satisfied by the training data *DB*s. The higher the weight scores, the larger the likelihood will be for the formulas. Thus, to maximize the likelihood, we try to maximize the score `WPLL` by choosing the proper weights of candidate formulas. We use a gradient-descent based algorithm to learn the weights to maximize the `WPLL` score, as shown in Algorithm 4.

Notice that, in Step 5 of Algorithm 4, x is selected from *DB*s by a random order. One specific random order corresponds to $|DBs|$ (the number of elements of *DB*s) repetitions from Steps 5 to 8, calculating a value of w^{i+1}. In our implementation, the final result of w^{i+1} is given by an average of the values attained according to five randomly generated orders. When computing `WPLL`, we count the true instantiations from one individual *DB*, instead of among different *DB*s (which can be found from Steps 5–9 of Algorithm 4). Each time after computing `WPLL`, we will update the weight w^{i+1} and use another *DB* to compute `WPLL` and do another updating of w^{i+1}, and so forth. In this way, the same objects or state indices among different plan traces will not affect the counting of true instantiations (since we do the counting only from one individual plan trace, instead of among different plan traces).

Algorithm 4 Weight learning algorithm: *learn_weights*(F, *DBs*, N).

```
Input: a list of formulas $F$, A list of databases $DBs$, the number of iterations $N$.
Output: Weights of candidate formulas $F$.
1: Initialize $w^0 = (0, \dots, 0)$.
2: set the number of iterations as $N$.
3: for $i = 0$ to $N - 1$ do
4:     $w^{i+1} = w^i$.
5:     for Each database $x$ in $DBs$ do
6:         Calculate WPLL$(w^{i+1}, x)$.
7:         $w^{i+1} = w^{i+1} + \lambda \cdot \frac{\partial \text{WPLL}(w^{i+1}, x)}{\partial w^{i+1}}$.
8:         (Notice that $\lambda$ is a small enough constant.)
9:     end for
10: end for
11: return $w^N$.
```

Since *DB*s are attained by Step 1 and formulas F are generated by F1–F5 in Step 2, by using Algorithm 4, we can learn the weights of the formulas F. Example 6 in the following will demonstrate the weight learning procedure.

Example 6. For simplicity, we use the closed world assumption in this example. Thus, in Table 3, *DB*2 can be denoted as {(is-at home 0), (at o1 l1 0), (move home l1 0), (put-in o1 l1 1), (in o1 2), (move l1 home 2), (is-at home 3), (at o1 home 3)} (other propositions not shown here are viewed as *false*), where $\{0, 1, 2, 3\}$ stands for $\{s_0, s_1, s_2, s_3\}$. Then x can be denoted as $\{\dots, 1, \dots, 1, \dots, 1, \dots\}$, where $x_i = 0$ is not shown here, which means its corresponding proposition does not appear in *DB*2, and $x_i = 1$ means its corresponding proposition appears in *DB*2. We assume that there are two locations {home, l1}, one portable {o1}, and four states $\{s_0, s_1, s_2, s_3\}$. Then, the number of propositions $n = |x|$ is 48, which can be calculated by counting all the groundings of {(is-at ?l ?i), (at ?o ?l ?i), (in ?o ?i), (move ?m ?l ?i), (put-in ?o ?l ?i), (take-out ?o ?i)}. Notice that we use a new parameter "?i" to denote states in each literal. Take the candidate formula "$\forall i.p.$(take-out o i) $\rightarrow$ (at o p i)" as an example, assuming that there is only one formula in an MLN. We denote the all groundings that contain the proposition (take-out o1 0) as $F_{(\text{take-out } o1\ 0)}$, which is $F_{(\text{take-out } o1\ 0)} = \{(\text{take-out } o1\ 0) \rightarrow (\text{at } o1\ l1\ 0), (\text{take-out } o1\ 0) \rightarrow (\text{at } o1\ home\ 0)\}$. Likewise, we can calculate $F_{(\text{take-out } o1\ 1)}$, $F_{(\text{take-out } o1\ 2)}$, $F_{(\text{take-out } o1\ 3)}$, $F_{(\text{at } o1\ l1\ 0)}$, $F_{(\text{at } o1\ l1\ 1)}$, $F_{(\text{at } o1\ l1\ 2)}$, $F_{(\text{at } o1\ l1\ 3)}$, $F_{(\text{at } o1\ home\ 0)}$, $F_{(\text{at } o1\ home\ 1)}$, $F_{(\text{at } o1\ home\ 2)}$, $F_{(\text{at } o1\ home\ 3)}$. When X_l is (take-out o1 0), and F_l is $F_{(\text{take-out } o1\ 0)}$, from

$$C_{(X_l=x_l)} = \exp \sum_{f_i \in F_l} w_i f_i\big(X_l = x_l, MB_x(X_l)\big)$$

we have

$$C_{((\text{take-out } o1\ 0)=0)} = e^{2w_i}$$
$$C_{((\text{take-out } o1\ 0)=1)} = e^{w_i}$$

where w_i is the weight of the candidate formula. Note that in the world state *DB2*,

$$f_i\big((\text{take-out } o1\ 0) = 0, MB_x\big((\text{take-out } o1\ 0)\big)\big) = 1$$

and

$$f_i\big((\text{take-out } o1\ 0) = 1, MB_x\big((\text{take-out } o1\ 0)\big)\big) = 0$$

when f_i is (take-out o1 0) $\rightarrow$ (at o1 l1 0), likewise for other f_i from $F_{(\text{take-out } o1\ 0)}$, $F_{(\text{take-out } o1\ 1)}$, $F_{(\text{take-out } o1\ 2)}$, $F_{(\text{take-out } o1\ 3)}$, $F_{(\text{at } o1\ l1\ 0)}$, $F_{(\text{at } o1\ l1\ 1)}$, $F_{(\text{at } o1\ l1\ 2)}$, $F_{(\text{at } o1\ l1\ 3)}$, $F_{(\text{at } o1\ home\ 0)}$, $F_{(\text{at } o1\ home\ 1)}$, $F_{(\text{at } o1\ home\ 2)}$, or $F_{(\text{at } o1\ home\ 3)}$. Finally, we can calculate WPLL(w, x) by

$$\begin{aligned}\text{WPLL}(w, x) &= \sum_{l=1}^{n} \log \frac{C_{(X_l=x_l)}}{C_{(X_l=0)} + C_{(X_l=1)}} \\ &= \log \frac{e^{2w_i}}{e^{2w_i} + e^{w_i}} + \log \frac{e^{2w_i}}{e^{2w_i} + 1} + \log \frac{e^{2w_i}}{e^{2w_i} + 1} + \log \frac{e^{2w_i}}{e^{2w_i} + e^{w_i}} + 8 \log \frac{e^{w_i}}{e^{w_i} + e^{w_i}}\end{aligned}$$

where the first four items are attained when $X_l =$ (take-out o1 s_0), (take-out o1 s_1), (take-out o1 s_2), (take-out o1 s_3), respectively; the last item is attained when $X_l =$ (*at* o1 l1 s_0), (*at* o1 l1 s_1), (*at* o1 l1 s_2), (*at* o1 l1 s_2), (*at* o1 l1 s_3), (*at* o1 *home* s_0), (*at* o1 *home* s_1), (*at* o1 *home* s_2), (*at* o1 *home* s_2), (*at* o1 *home* s_3), respectively. Note that w_i is the weight of the candidate formula "$\forall i.p.$(take-out o i) $\rightarrow$ (at o p i)", and w is the same as w_i since there is only one candidate formula in the MLN. By setting the weight w with an initial value, we can calculate w iteratively using Algorithm 4.

As an example, the learning result for the formulas in Table 4 is shown in Table 10. Intuitively speaking, the larger the weight of a formula, the more probable that the formula is true in the world state.

Table 10
An example of the learned weights of formulas generated by F1.

ID	Weights	Formulas
1	-0.5	$\forall i.p.((\text{take-out } o\ i) \rightarrow (\text{at } o\ p\ i))$
2	-1.2	$\forall i.x.p.((\text{take-out } o\ i) \rightarrow (\text{at } x\ p\ i))$
3	-1.5	$\forall i.p.((\text{take-out } o\ i) \rightarrow (\text{is-at } p\ i))$
4	0.8	$\forall i.((\text{take-out } o\ i) \rightarrow (\text{in } o\ i))$
5	-0.9	$\forall i.x.((\text{take-out } o\ i) \rightarrow (\text{in } x\ i))$

Table 11
The generated action model.

action:	take-out(?x - portable)
preconditions:	(in ?x)
	...
effects:	...

Table 12
The formulas selected from Tables 5, 7 and 8.

ID	Selected formulas
1	$\forall i.((\text{take-out } o\ i) \wedge \rightarrow (\text{in } o\ i))$
2	$\forall i.((\text{take-out } o\ i) \rightarrow \neg(\text{in } o\ i+1) \wedge (\text{in } o\ i))$
3	$\forall i.p.((\text{take-out } o\ i) \wedge (\text{is-at } p\ i) \rightarrow \neg(\text{at } o\ p\ i) \wedge (\text{at } o\ p\ i{+}1))$

4.4. Step 4: generating action models

All weights in MLN learning are initialized to zero. The optimization of pseudo log-likelihood ensures that when the number of true groundings of f_i is larger, generally, the corresponding weight of f_i will be higher. Hence, the final weight of a formula in an MLN is a confidence measure of that formula. Intuitively, the larger the weight of a formula, the more probable that formula will be true in the world description. However, when generating the final action model from these formulas, we need to determine a threshold δ, based on the accuracy of action models learned to choose a set of formulas from an MLN. We will describe the steps of generating action models in Algorithm 5.

Algorithm 5 Generate action models: *attain_models*(W, F).

```
Input: A set of candidate formulas $F$ and their weights $W$.
Output: A set of action models $A$.
1: Initialize $A = \emptyset$;
2: Test and choose a threshold value $\delta$ based on the error estimates of the plan correctness (see Section 5.1) in the evaluation criteria using the training plan traces;
3: Select all the formulas $F' \in F$ whose corresponding weights in $W$ are larger than $\delta$;
4: Convert $F'$ to action models $A$ based on F1–F5;
5: return $A$;
```

For instance, if a formula generated by F1 is selected, the predicate p in the formula will be transformed to a precondition of the action a in the formula. The action model generation process can be seen in Example 7.

Example 7. From the result of Example 6, if we set zero as the threshold, we can select the formulas whose weights are larger than zero from Table 10. The result is: "$\forall i.(\text{take-out } o\ i) \rightarrow (\text{in } o\ i)$", whose weight is 0.8. After converting the formula to an action model, we get the result shown in Table 11, where the ellipsis represents the preconditions or effects learned by other formulas but not in Table 10.

Furthermore, to demonstrate how to generate a precondition, a positive/negative effect or a positive/negative conditional effect from a selected formula (whose weight is larger than the threshold δ), we assume that there is one formula selected from each of Tables 4, 7 and 8 respectively, which is shown in Table 12. We then convert the formulas to the corresponding action model as shown in Table 13. The ID numbers in Tables 12 and 13 indicate their corresponding conversion relation.

From Table 13, a precondition "(in ?o)", can be attained by converting the first formula in Table 12. This precondition means that if we want to take out a portable "?o" from the briefcase, the portable "?o" should be in the briefcase. Likewise, we can give an explanation for the effects in Table 13.

Notice that we do not show how to generate a precondition of the action "take-out" in an implication form, since such a precondition does not exist in the real action model of "take-out". However, for the domain *trucks* whose learning result is shown in Table 20 in the experiment section, a precondition of the action "load(?p-package ?t-truck ?a1-truckarea ?l-location)" in an implication form

Table 13
The generated action model according to Table 12.

action:	take-out(?o - portable)	ID
preconditions:	(in ?o)	1
effects:	(and (not (in ?o))	2
	(forall ?p - location (when (is-at ?p)(at ?o ?p)))	3

$$\text{“}\big(\text{forall (?a2-truckarea)}\big(\text{imply(closer ?a2 ?a1) (free ?a2 ?t)}\big)\big)\text{”}$$

will be learned from the candidate formula “$\forall i.a2.((\text{load } p\ t\ a1\ l\ i) \wedge (\text{closer } a2\ a1\ i) \rightarrow (\text{free } a2\ t\ i))$” (i.e., its weight is high enough to be selected by LAMP), which is generated from **F1**. This precondition suggests that, in order to load a package “?p” into the area “?a1” of the truck “?t”, any other area “?a2” of “?t” close to “?a1” should be free.

4.5. Properties of action models

4.5.1. Action soundness and plan consistency

With the candidate formulas generated by **F1**–**F5** and the constraints **A1**–**A3**, we would like to show that certain properties are satisfied by our LAMP algorithm.

Soundness Consider a set of action models. We say that these actions are *sound* if whenever we apply a ‘legal chain’ of action sequence to a consistent initial state, such that all preconditions of actions in the sequence are satisfied in their preceding states, the resulting end state is consistent; here a state is considered *consistent* if it does not contain both a literal p and its negation ($\neg p$). A ‘legal’ action means that all preconditions of that action are satisfied in the state it is applied to.

Definition 1 *(Soundness of action models).* An action model is said to be *sound* if starting from any initial state S, any sequence P built by using a legal forward chaining of the action models leads to a consistent state.

Theorem 1 *(Soundness property). Imposing constraint* **A1** *on the candidate formulas in* F1–F5 *ensures that the action models learned are sound.*

Proof. This can be proven by induction. First, the initial state S is consistent. Assume that an action a is applicable to S. Consider the state T after a is applied to S. Then, suppose that T is inconsistent, that is, p and $\neg p$ must be both true in T. There are two cases. In case one, p is true in S and $\neg p$ is true in T, and p is not removed from T after a's execution. However, this would not be possible due to the semantics of PDDL. In the second case, p and $\neg p$ are both on the right-hand side of some rules in the effect of a, since there are no other domain axioms that can infer p and $\neg p$. Furthermore, the left-hand side of these rules L_1 and L_2 are both true in T. This means that for the rules $L_1 \rightarrow p$ and $L_2 \rightarrow \neg p$ that are both in effects of a, L_1 and L_2 are both true. However, we know from the constraint **A1** that this is not allowed. Thus, T must be consistent. By induction, any forward-chaining sequence must lead to a consistent state only. □

Above we have shown that our learned action models are sound. A related question is *completeness*; that is, whenever a plan exists for a planning problem, a planner can generate at least one solution to the problem. However, when generating action models, we cannot guarantee that the models are ‘complete’, since completeness is a property of both a planning system and an action model together, rather than an action model alone. Thus, we will not consider completeness for our action models.

Next, we wish to show that our learned action models are sufficiently expressive to ‘explain’ all the training plan traces. This means that using our action models in the same order of a plan trace in the training set, we can obtain the goal conditions from the initial conditions. This is known as plan consistency property.

Definition 2 *(Plan consistency).* We denote a plan trace as $\{s_0, a_0, s_1, \ldots s_n, a_n, g\}$, where s_i is a state before the execution of the action a_i, and denote the corresponding action model of a_i as $M(a_i)$. We say the learned action models are *consistent* with a plan trace if and only if the following two conditions are satisfied: for each action a_i in the plan trace, where $0 \leqslant i \leqslant n$,

1. all the preconditions of $M(a_i)$ are satisfied in state s_i; and
2. the goals g can be achieved by executing the action sequence.

Theorem 2 *(Plan consistency). The action models learned by LAMP are* consistent *with the training plan traces.*

Table 14
The size of problems.

Domains	Predicates	Actions	Plan traces	Average length
Briefcase	3	3	100	13
Elevator	6	3	180	19
Openstacks	9	5	200	31
Trucks	10	4	180	26

Proof. According to Definition 2, we only need to verify the two conditions of *consistency* are satisfied. From constraint **A2**, we know that all preconditions of every action a_j in a plan trace will be either added by some action a_i prior to it, or exist in the initial state, and should not be deleted until they are used as preconditions of the action; i.e., they are all satisfied. Furthermore, every goal literal in the goal state will be either added by some action prior to the goal state, or exist in the initial state; i.e., the goal can be achieved by executing the action sequence. Thus, from the definition of *consistency*, it follows that the conclusion holds. □

4.5.2. Complexity analysis

Comparing our LAMP algorithm with two previous algorithms, ARMS [52] and SLAF [39], in learning action models, we note that ARMS uses weighted-MAXSAT to learn STRIPS models, which are action models with no quantifier in preconditions or effects. SLAF imposes quantifiers on its learning result, in particular, existential quantifiers appear in preconditions and universal quantifiers appear in effects. Such a constraint is rather arbitrary and strong for many real-world action domains. Our algorithm, LAMP, learns quantifiers in a way which conforms to the definition of PDDL, such that preconditions can be quantified by universal quantifiers, effects that are literals can be quantified by universal quantifiers and effects that are conditional effects can be quantified by both existential and universal quantifiers on their condition part, and universal quantifiers on their effect part. Besides, LAMP learns models with implications as preconditions, which makes the learned action models more expressive.

We now analyze the time complexity of the LAMP algorithm. The running time of the LAMP algorithm depends on the running time of each step in the algorithm. In the first step, the running time is $O(tlg)$, where t denotes the number of plan traces, l denotes the maximum length of plan traces. and g denotes the maximal number of propositions in states including intermediate states, initial states and goal states. In the second step, the running time is $O(ap^n)$, where n is the maximal number of predicates to form a conditional effect or an implication formula, p and a represent the number of predicates and the number of actions, respectively. It takes $O(mntlgf)$ in the third step, where f denotes the number of formulas, m denotes the number of iterations. Finally, it takes $O(f)$ in the fourth step. Thus, the total running time of LAMP is $O(tlg) + O(ap^n) + O(mntlgf) + O(f) = O(mntlgf)$, where ap^n is generally much smaller than $mntlgf$. Likewise, the space complexity of the fourth steps is $O(tlg)$, $O(ap^n)$, $O(ntlgf)$ and $O(f)$ respectively. Thus, the space complexity of LAMP is $O(tlg) + O(ap^n) + O(ntlgf) + O(f) = O(ntlgf)$, since ap^n is generally much smaller than $O(ntlgf)$.

Comparing the computational complexity, we note that the complexity of SLAF is $O(sk(2|P|)^{k+1})$, where s is equivalent to tl, k is the minimum number such that preconditions are in k-DNF form and P is a set of all possible fluents. P is larger than g and f. Thus, the time complexity of LAMP is lower than that of SLAF if m is assumed to be a constant.

5. Experiments

5.1. Datasets and evaluation criteria

To evaluate LAMP, we collected plan traces from the following planning domains: *briefcase, elevator, openstacks, trucks*, where *briefcase* was from the homepage of IPP,[1] *elevator* was from the second International Planning Competition (IPC-2),[5] *openstacks* and *trucks* were both from the fifth International Planning Competition (IPC-5). These domains have the characteristics we need to evaluate in our LAMP algorithm: *briefcase* and *elevator* domains have quantified conditional effects, *openstacks* and *trucks* both have quantifiers and implications in preconditions. Using the FF planner,[6] we generated 100 plan traces from *briefcase*, 180 plan traces from *elevator*, 200 plan traces from *openstacks* and 180 plan traces from *trucks* by solving planning problems from each domain. Notice that the number of plan traces depends on the number of planning problems we downloaded from IPP, IPC-2 and IPC-5. We show the size of the learning problems in our experiment in Table 14, where the second and third columns are the numbers of different predicates and actions in each domain; the fourth column is the number of plan traces collected from each domain; the last column is the average length of plan traces in each domain.

We use all the plan traces as training data to learn action models, and use the corresponding hand-written action models from IPP, IPC-2 or IPC-5 as the *ground truth action models* to compare with our learned action models. The comparison gives us the error rates. This method of evaluation allows us to generate action models using our algorithm and then compare the

[5] http://www.cs.toronto.edu/aips2000/.
[6] http://members.deri.at/~joergh/ff.html.

results to the hand-crafted ones in the IPC-5 collection. The accuracy thus obtained gives us assurances of the effectiveness of the action models we learn in the real world, when we only have the observed states and activity sequences from sensors.

We define the error rates of our learning algorithm as the difference between our learned action models and the hand-written action models that are considered as “ground truth”. If a precondition appears in the precondition list of our learned action model but not in the precondition list of its corresponding hand-written action model, the error count of preconditions, denoted by E_{pre}, increases by one. If a precondition appears in the precondition list of a hand-written action model but not in the precondition list of the corresponding learned action model, E_{pre} also increases by one. Likewise, the error count in the actions' effects is denoted as E_{eff}. Different quantifiers and implications both account for a difference of one.

Note that the preconditions of an action are conjunctive, which are viewed as a set, where each element is an implication or a universally quantified literal. We denote the set of all the possible preconditions of an action model as L^0_{pre}, the set of preconditions of the learned action model as L^1_{pre}, and the set of preconditions of the ground truth action model as L^2_{pre}. Then, the error count of preconditions is calculated by comparing L^1_{pre} and L^2_{pre}, i.e. $E_{\text{pre}} = |L^1_{\text{pre}} \cup L^2_{\text{pre}} - L^1_{\text{pre}} \cap L^2_{\text{pre}}|$.

Likewise, the effects of an action model are also conjunctive, which are viewed as a set, each element of which is a universally quantified literal, or a conditional effect. We denote the set of all the possible effects of an action model as L^0_{eff}, the set of effects of the learned action model as L^1_{eff}, and the set of effects of the ground truth action model as L^2_{eff}. Then the error count of effects is calculated by $E_{\text{eff}} = |L^1_{\text{eff}} \cup L^2_{\text{eff}} - L^1_{\text{eff}} \cap L^2_{\text{eff}}|$.

Furthermore, we denote the total number of all the possible preconditions and effects of an action model as T_{pre} and T_{eff}, respectively. Then we have $T_{\text{pre}} = |L^0_{\text{pre}}|$ and $T_{\text{eff}} = |L^0_{\text{eff}}|$. Here we constrain implications and conditional effects with only a constant number of different literals, so that T_{pre} and T_{eff} are both finite.

In our experiments, the error rate of an action model a is defined as

$$R(\mathrm{a}) = \frac{1}{2}\left(\frac{E_{\text{pre}}}{T_{\text{pre}}} + \frac{E_{\text{eff}}}{T_{\text{eff}}}\right)$$

where we assume the error rates of preconditions and effects were equally important, and the range of error rate $R(a)$ is within $[0, 1]$. Furthermore, the error rate of all the action models A in a domain is defined as

$$R(A) = \frac{1}{|A|}\sum_{a \in A} R(a)$$

where $|A|$ is the number of A's elements. Using this definition of error rate, we present our experimental results in the next subsection. We will evaluate LAMP with respect to the following criteria: (1) the relationship between accuracy and the percentage of observed intermediate states, (2) the relationship between accuracy and the percentage of propositions in each state, (3) the relationship between accuracy and the number of plan traces in action-model learning, (4) the running time, (5) the human effort saved by LAMP, (6) the application of LAMP in software requirement engineering, and (7) the example output. The detailed description of each criterion is given in the next subsection.

5.2. Experimental results

5.2.1. Relationship between accuracy and the percentage of observed intermediate states

To simulate partial observation between two actions in a plan trace, from the plan traces, we randomly selected observed states with specific percentage of observations $\frac{1}{5}$, $\frac{1}{4}$, $\frac{1}{3}$, $\frac{1}{2}$, and 1. For each percentage value, e.g., $\frac{1}{5}$, and likewise for other percentage values, we randomly selected an observation within five consecutive states in a plan trace. We ran the selection process five times. Each time our LAMP algorithm generated the learned action models, from which we calculated an error rate. Finally, we calculated an average error rate on the plan traces. The results of these tests are shown in Fig. 1.

Fig. 1 shows the performance of the LAMP algorithm with respect to different threshold values δ used in selecting the candidate formulas in the last step of algorithm, which were set to 0.01, 0.1, 0.5 and 1.0, respectively. From the results, we find that the performance is sensitive to the choice of the threshold; the threshold values should neither be too large nor too small. A threshold that is too large may miss useful candidate formulas, and a threshold that is too small may bring in too many noisy candidate formulas that affect the overall accuracy of the algorithm. In these experiments, it can be seen that when the threshold is set as 0.5, the mean average accuracy is optimal (Fig. 1(II)). Furthermore, the error bars, which show the confidence intervals, show that our algorithm performance is stable.

We also would like to know the relationship between the accuracy of the learned model and the percentage of observed intermediate states. Since the LAMP algorithm does not require all of the intermediate state information to be seen, it can still learn useful information from some but not all observations. In our experiment, we chose different percentages of observations: $\frac{1}{5}$, $\frac{1}{4}$, $\frac{1}{3}$, $\frac{1}{2}$, 1, and produced the corresponding error-rate results. Our experimental results, which are given in Fig. 1, show that in most cases, the more observations that we have, the lower the error rate will be, which is consistent with our intuition. However, there are some cases, e.g., when the threshold δ is set to be 1.0, and there are only $\frac{1}{3}$ of the states observed, the error rate is lower than the case when $\frac{1}{2}$ of the states are given. These cases are not consistent with our intuition, but they are possible, however, since when more observations are obtained, the weights of their corresponding

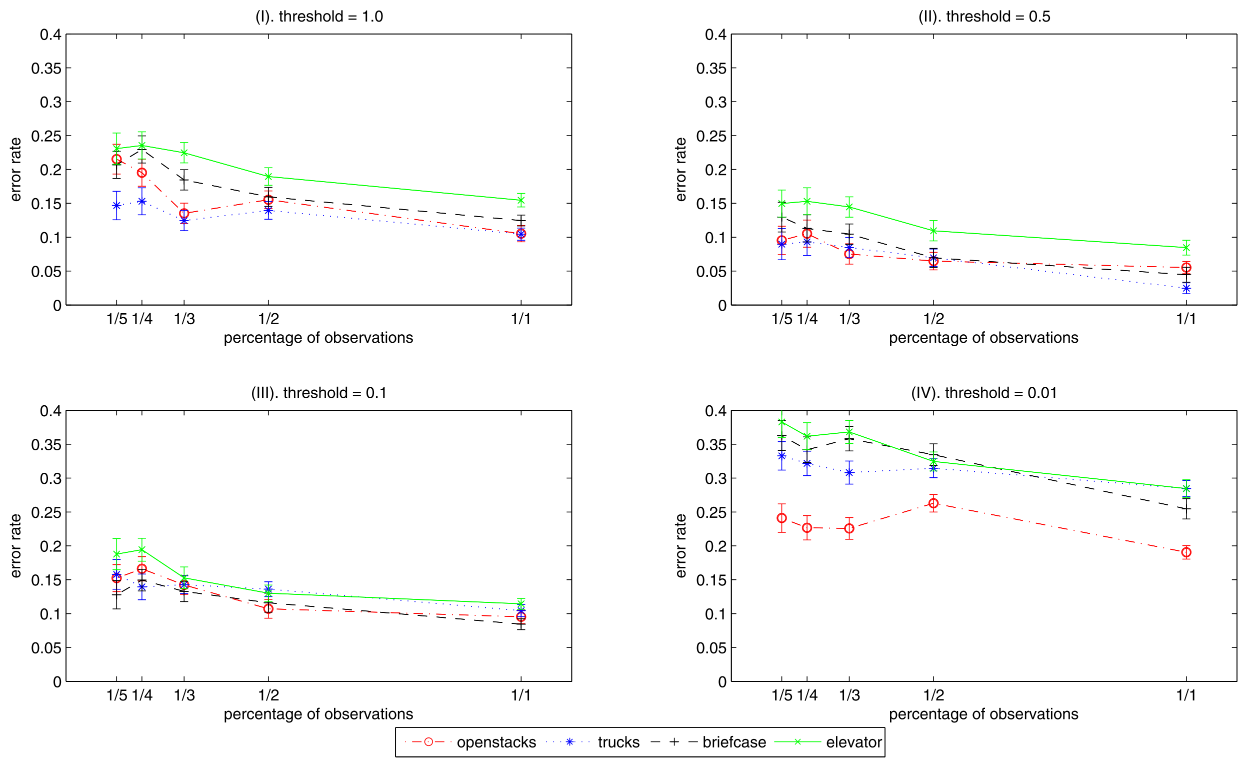


Fig. 1. The error rates with respect to different percentages of observations.

formulas go up and the weights of other formulas may go down when considering the overall learning process. Thus, if the threshold δ is still set to be 1.0, some formulas which were chosen before are missed out, and the error rate will be higher. Thus, we conclude that in these cases, we need to reduce the value of the threshold correspondingly for the error rate to decrease.

5.2.2. Relationship between accuracy and the percentage of propositions in each state

Another aspect of partial observations was simulated by reducing the percentage of propositions known to be true in each state. In this section, we performed tests on this aspect of partial observation.

We first set the percentage of observations as $\frac{1}{3}$, and tested different percentages of propositions in each state to calculate their corresponding errors. The propositions in each state were randomly selected for each specific percentage value to generate the observations. The action models were then learned and evaluated on the ground truth models. The results are shown in Fig. 2, where a value 20 in the x-coordinate means 20% of propositions in each state are given in a plan trace, likewise for 40, 60, 80.

From Fig. 2 we find that, on one hand, when setting the candidate-selection threshold value δ to 0.5 (shown in Fig. 2(II)), the error rate is generally lower than using other thresholds (shown in Fig. 2 (I), (III) and (IV)), which is consistent with the results of Fig. 1. On the other hand, when the percentage of propositions increases, the error rate generally decreases. This is explained as: the larger the percentage is, the more information available will be attained by our learning algorithm, which will help improve the learning result.

From Figs. 1 and 2, we find that the error rate of the domain *elevator* is generally larger than other domains, which suggests that it is more difficult to learn than the others. We observe that *elevator* has actions (e.g., stop), which contain more conditional effects than others. From **F4** (or **F5**), we know that conditional effects need more literals to represent them than other conditions (preconditions, positive and negative effects), which can be seen from **F1** to **F3**. This fact will make conditional effects difficult to learn. That is why the error rate of *elevator* is larger than the others. The similar result can also be found from *briefcase* whose conditional effects are less than *elevator* but more than *openstacks* and *trucks*. Its error rate is generally larger than that of *openstacks* and *trucks*, but smaller than that of *elevator*.

5.2.3. Relationship between accuracy and the number of plan traces in action-model learning

To see how the error rate was affected by the number of plan traces, we used different number of plan traces as the training data to evaluate the performance. In our tests, we assumed that each plan trace had $\frac{1}{5}$ of fully observed intermediate states. These observed states were randomly selected. The process of generating state observations was repeated five times, where each time an error rate was generated under different selections.

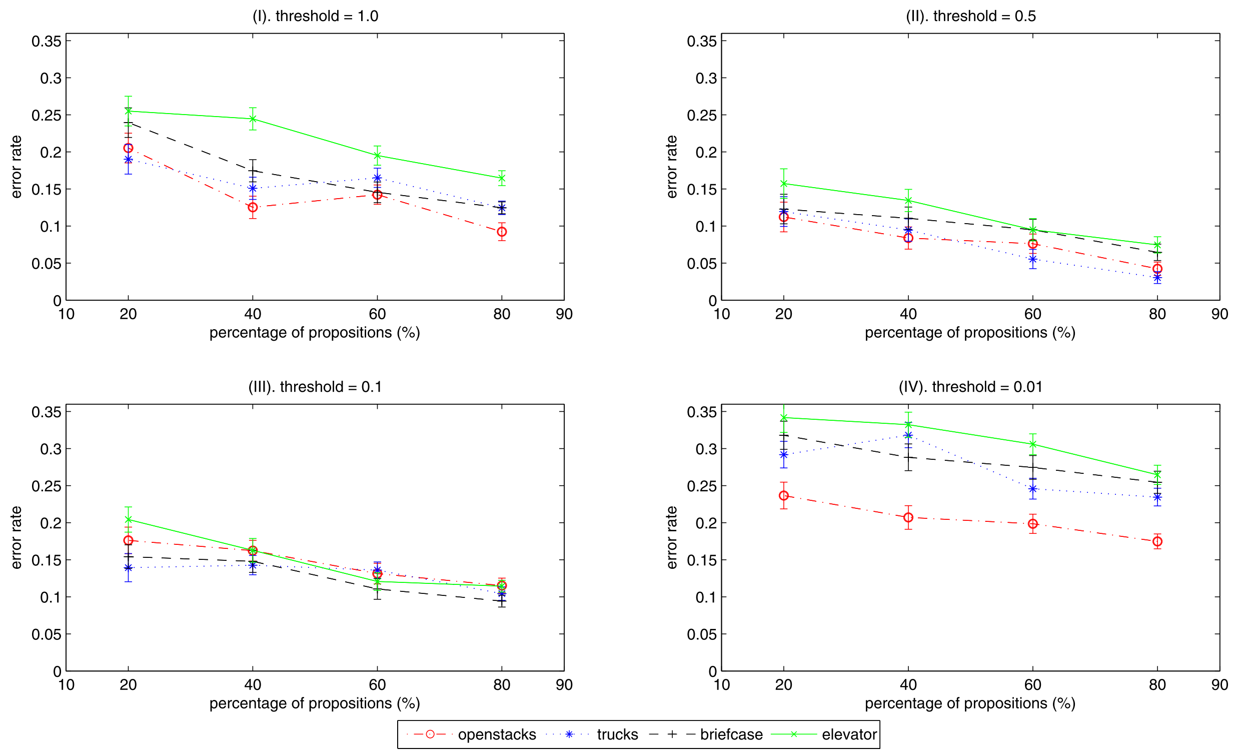


Fig. 2. The error rates with respect to different percentages of propositions in each state.

We show the results in Fig. 3, which suggests that the error rate generally decreases when the number of plan traces increases. At the beginning, the error rate decreases quickly but eventually it goes down slowly. This means that the error rate is more sensitive when the number of plan traces is small than when it is large. When comparing the different curves in a figure, we can see that the error rate is generally lower when the threshold δ is 0.5 than other thresholds, which is consistent with the result from Fig. 1.

5.2.4. *Running time*

We also tested the different number of plan traces to obtain the corresponding CPU time for learning. The results is shown in Fig. 4, where the percentage of observed intermediate states is set by $\frac{1}{5}$. From Fig. 4 (I)–(IV), we can see that, the CPU time goes up when the number of plan traces increases. To see the relationship between CPU time and the number of plan traces, we fit the running result of domain *briefcase* with a nonlinear curve, which is shown in Fig. 5. The functional form of the curve in Fig. 5 is $-0.0010x^3 + 0.0795x^2 + 14.7737x - 147.5333$, which means that the CPU time increases polynomially with respect to the number of plan traces. A similar result can also be found from three other domains: *elevator*, *openstacks* and *trucks*.

5.2.5. *Human effort saved by* `LAMP`

The next question is: if we use `LAMP` to create action models, how much human effort will be reduced as compared to not asking human experts to manually encode the domains from scratch? We invited two groups of people, each having a size of 10, to build action models. We took care to separate the two groups in our testing.

Case 1 Testers in the first group were not given any initial action models, so that they created action models from "scratch";

Case 2 Testers in the second group were given the action models learned by `LAMP`, and were asked to revise the models.

These two groups of testers consisted of people between 21 and 31 years of age. In the first group, nine people were male and one was female. In the second group, eight people were male and two were female. Among the ten people in the first group, six were students and faculties from universities and the remaining four were from companies. In the second group, seven people were from universities and three were from companies. All these people had some general knowledge on AI Planning and PDDL language. We were interested in seeing how much difference was there between the groups in the quality and human effort of action model construction.

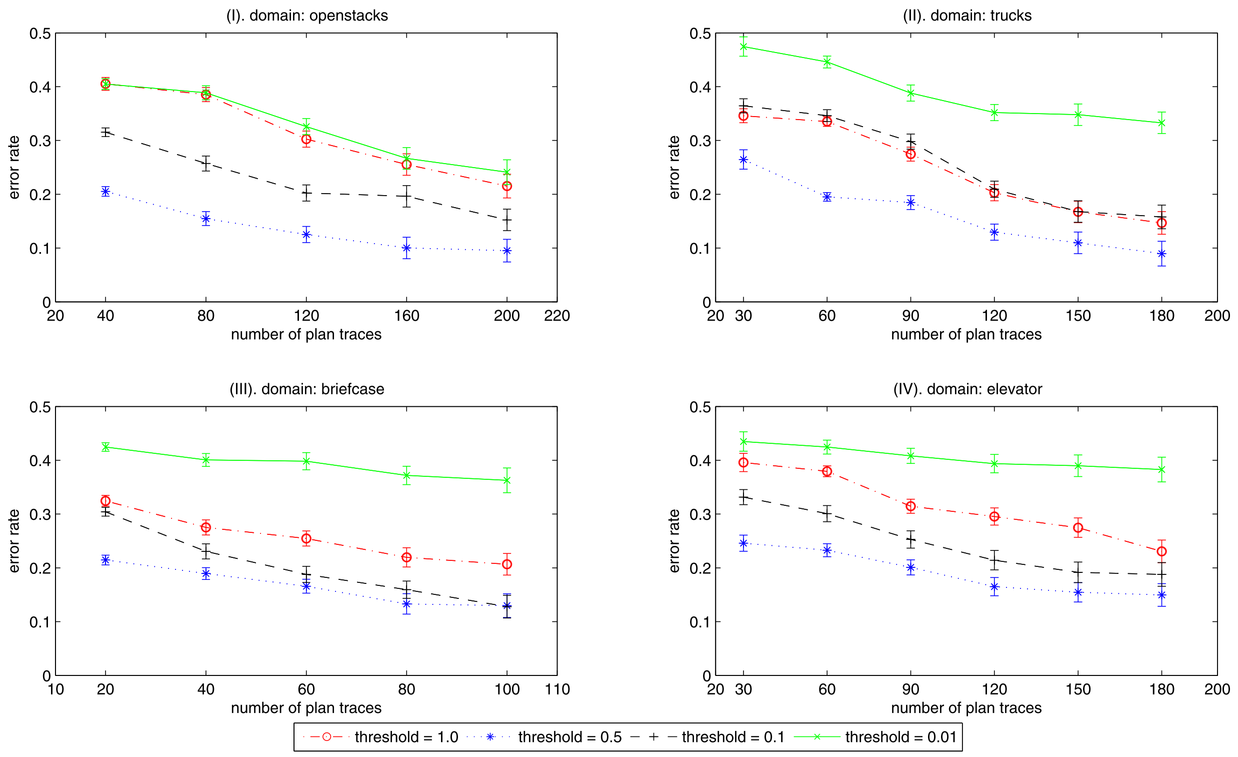


Fig. 3. The error rates with respect to different number of plan traces, with the percentage of observed intermediate states as $\frac{1}{5}$.

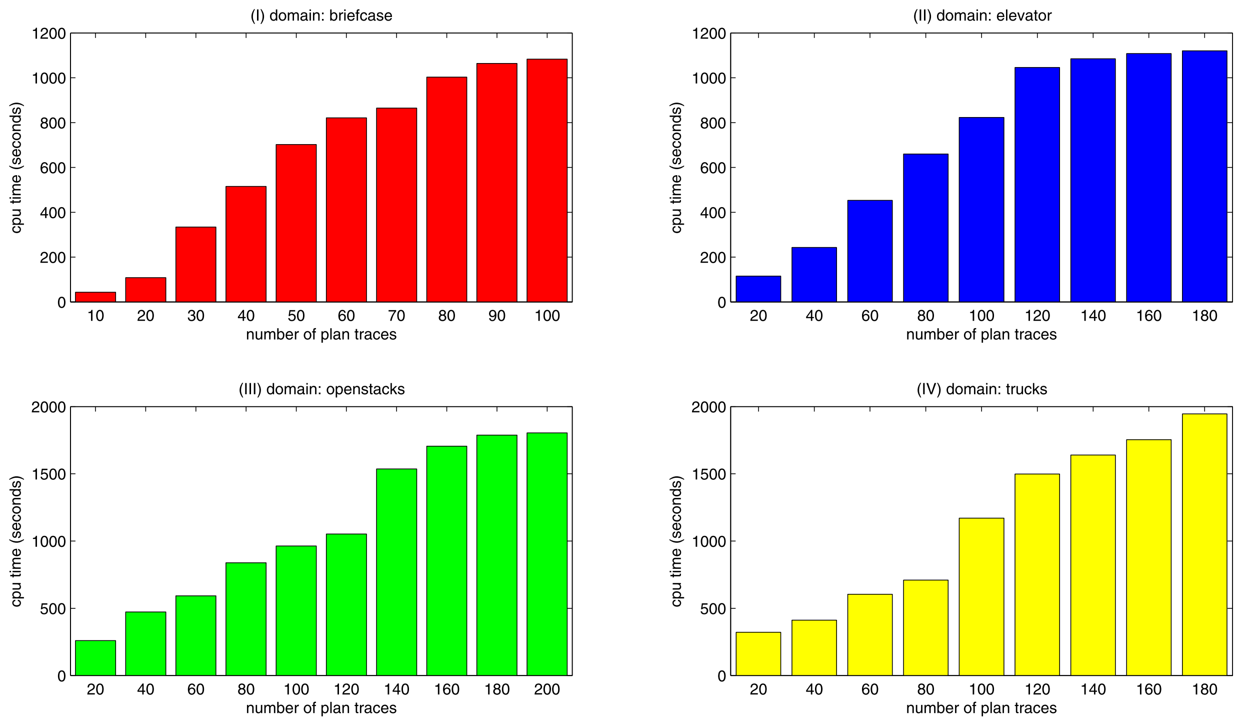


Fig. 4. The CPU time with respect to different number of plan traces.

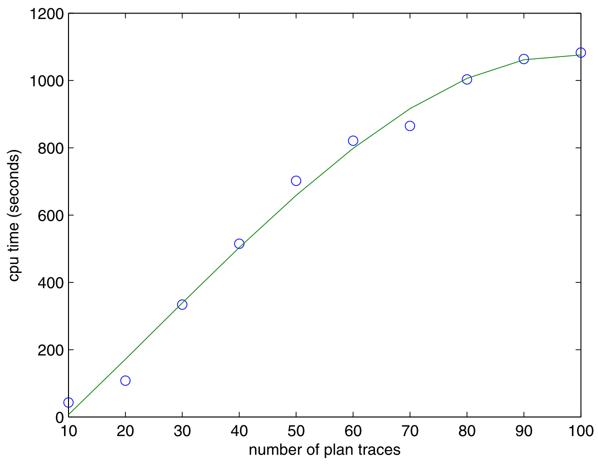


Fig. 5. The fitting result of the CPU time of *briefcase*.

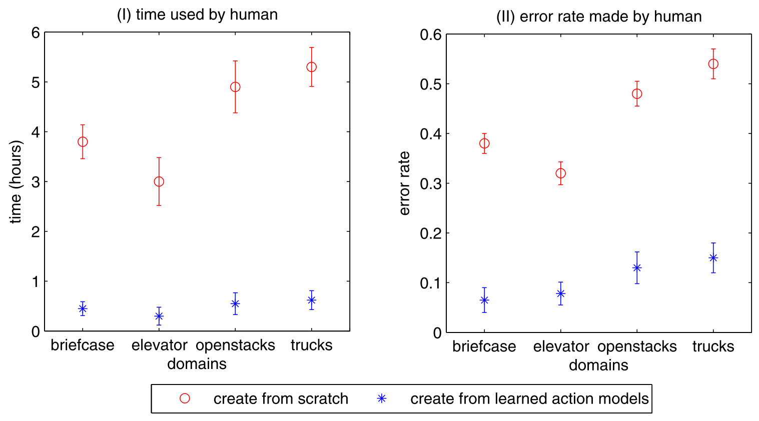


Fig. 6. Time saved and error rate reduced by LAMP.

For each test case, we recorded the testers' individual times spent on model building, and calculated their corresponding error rates as compared to the ground-truth models. We present our results in Fig. 6, where "$\circ$" indicates the testing results for the first case, and "$*$" indicates the ones for the second case. Notice that, the results as shown in Fig. 6 are the average results of ten people in each group. From Fig. 6(I), we find that, the time cost of the first case is much lower than that of the second case for all four domains *briefcase*, *elevator*, *openstacks* and *trucks*. Similar results on error rates can also be found in Fig. 6(II). In summary, from this experiment, we can see that our LAMP algorithm can indeed help reduce the human effort and improve the model quality in the construction of action models.

5.2.6. Comparison of grounded actions

In order to show that our method of error counting for learned actions with quantifiers and implications is reasonable, in this section we compare the error rates obtained with that of grounded actions. We test the differences between hand-crafted action models and learned action models by *grounding* quantifiers (or implications, conditional effects). The results are STRIPS action models. For instance, the action model "move" of Table 2 can be grounded in the initial state of plan trace 2 in Table 1 to the result as shown in the following,

(move home l1)	
preconditions:	(is-at home)
effects:	(and (is-at l1) (not (is-at home)))

where the quantifier and conditional effect are grounded, which results in a STRIPS action. We count the differences between grounded hand-crafted action models and grounded learned action models and generate error rates as described in Section 5.1. We no longer need to count the differences between quantifiers and conditional effects.

We learn action models by setting the threshold δ as 0.5 and the percentage of observed intermediate states as $\frac{1}{5}$. We wish to test differences between the learned action models and hand-crafted action models by *grounding* them in plan

Table 15
Error rates of actions with quantifiers ($R(A)$) and error rates of grounded actions ($R'(A)$).

	Briefcase	Elevator	Openstacks	Trucks
$R'(A)$	0.11	0.14	0.08	0.09
$R(A)$	0.13	0.15	0.10	0.09

traces. We collected 20 testing plan traces with fully observed intermediate states for each domain including *briefcase*, *elevator*, *openstacks* and *trucks*, and ground the learned action models and hand-crafted action models in the states. We then calculate an average error rate (denoted as $R'(A)$) over all the STRIPS actions. The result is as shown in Table 15.

In Table 15, $R'(A)$ gives the error rates generated with *grounding*, while $R(A)$ gives the error rates generated from the quantified actions. From Table 15 we can see that the error rate $R'(A)$ is generally smaller than $R(A)$. We can see from the table that both methods of error counting result in about the same values for all error rates. This justifies our previously used error rate calculation method when considering quantified actions.

5.2.7. Applying LAMP to software requirement engineering

To demonstrate the real-world application of our LAMP algorithm, we consider a problem to acquire *software requirement specification* for specifying an online-service business process, which can also be found from our previous work [58]. The software system based on actions learned with LAMP is operational. As an example, we have successfully applied it to a *Bookstore System* in Guangxi province of China. A *software requirement specification* is a complete description of the behaviors of the system to be developed. It includes a set of user cases that describe all the interactions that users would have with the system. In this section, our main idea of using LAMP to acquire *software requirement specification* is given as follows.

1. We first extracted *types* that objects in the system belonged to, *predicates* that represented the relations among *types*, and *action schemas* that represented behaviors of the system.
2. After that, we collected *plan traces* from various business process by communicating with a business personnel.
3. Finally, we learned *action models* with *predicates, action headings and plan traces* as input, and converted the learned result to a *software requirement specification*.

In the following, we present an example that LAMP is used to acquire the *software requirement specification* of the *Bookstore Ordering System*.

Example 8. A Bookstore Ordering System can be separated by three levels:

1. Client ordering: a client orders his books from a base store based on a list of books, and generates an initial_ordering_form.
2. Base store ordering: a staff of the base store orders his books from a province store based on initial ordering forms, and generates a base_ordering_form.
3. Province store ordering: a staff of the province store orders his books from a book supplier, and generates a province_ordering_form.

The *types, predicates* and *action headings* extracted from the Bookstore Ordering System are shown as follows.

```
(:types
:   staff form department - object
:   lbook ioform boform poform - form
:   bstore pstore sdep - department
)
(:predicates
:   (init ?x - form)    // the form ?x is in initial state
:   (waiting ?x - form)    // the form ?x is in waiting state
:   (dealt ?x - form)    // the form ?x is dealt
:   (free ?x - staff)    // the staff ?x is free
:   (busy ?x - staff)    // the staff ?x is busy
:   (doing ?x - staff ?y - form)    // the staff ?x is dealing with the form ?y
:   (belong ?x - staff ?y - department)    // the staff ?x belongs to department ?y
)
(:action headings
:   (ClientOrder ?x - staff ?y - bstore ?a - ioform)
:   (BaseOrder ?x - staff ?y - pstore ?a - boform)
:   (ProvinceOrder ?x - staff ?y - sdep ?a - poform)
:   (OrderDone ?x - staff ?a - form ?b - form)
)
```

Table 16
Action models of Bookstore Ordering System.

(ClientOrder ?x - staff ?y - bstore ?a - ioform)	
precondition:	(free ?x) (init ?a) (belong ?x ?y)
effect:	(:and (busy ?x) (not (free ?x)) (not (init ?a)) *(doing ?x ?a)* (forall ?b - lbook (when (waiting ?b) (and (doing ?x ?b)(not (waiting ?b))))))
(BaseOrder ?x - staff ?y - pstore ?a - boform)	
precondition:	(free ?x) (init ?a) *(doing ?x ?a)* (belong ?x ?y)
effect:	(:and (busy ?x) (not (free ?x)) **(not (init ?a))** (forall ?b - ioform (when (waiting ?b) (and (doing ?x ?b)(not (waiting ?b))))))
(ProvinceOrder ?x - staff ?y - sdep ?a - poform)	
precondition:	(free ?x) (init ?a) (belong ?x ?y)
effect:	(:and (busy ?x) (not (free ?x)) (not (init ?a)) (forall ?b - boform (when (waiting ?b) (and (doing ?x ?b)**(not (waiting ?b))**)))))
(OrderDone ?x - staff ?a - form ?b - form)	
precondition:	*(dealt ?b) (waiting ?a)* (busy ?x) (doing ?x ?a) (doing ?x ?b)
effect:	(:and (dealt ?a) (waiting ?b) (free ?x) *(not (dealt ?b))* (not (busy ?x)) **(not (doing ?x ?a)) (not (doing ?x ?b))**)

where *staff, form* and *department* are the *types* of a ‘staff’, ‘form’, and ‘department’ in the Bookstore Ordering System; *lbook, ioform, boform* and *poform* are the *types* of ‘a list of book’, ‘an initial ordering form’, ‘a base ordering form’, and ‘province ordering form’ respectively; *bstore, pstore*, and *sdep* are the *types* of ‘base store’, ‘province store’ and ‘supplier’s department’ respectively. Corresponding to three business processes *client ordering, base store ordering and province store ordering*, there are three action headings *ClientOrder*, *BaseOrder* and *ProvinceOrder*. Since executing these actions is a process, we need an action to stop the process which we call *BookDone*.

Next, we collected plan traces from the book ordering domain. An example of a plan trace is shown as follows. Notice that a plan trace we use is only composed of an initial state, an action sequence and a goal, without any intermediate state. This would save us much time on collecting plan traces.

(:init
(belong staff1 basestore1) (belong staff2 provincestore1) (belong staff3 sdep1) (waiting lbook1) (init ioform1) (init boform1) (init poform1) (free staff1) (free staff2) (free staff3)
)
(:actions
: *(ClientOrder staff1 basestore1 ioform1)*
: *(OrderDone staff1 lbook1 ioform1)*
: *(BaseOrder staff2 provincestore1 boform1)*
: *(OrderDone staff2 ioform1 boform1)*
: *(ProvinceOrder staff3 sdep1 poform1)*
: *(OrderDone staff3 boform1 poform1)*
)
(:goal
(dealt lbook1) (dealt ioform1) (dealt boform1) (waiting poform1)
)

With the extracted *predicates, action headings* and 20 *plan traces* as input, we ran our LAMP algorithm to learn action models. We show an example of the running result in Table 16, where the part which is italicized and not emphasized means it should be deleted and the part which is emphasized means it should be added. This action model ‘ClientOrder’ means that, the process of client ordering can be executed only when the staff ?*x* who belongs to a base store was free, and the initial ordering form ?*a* is true in the initial state. As a result of the execution, the staff ?*x* become busy, and all the *book lists* ?*b* are being dealt with if they are in the state of *waiting*. Likewise for the statements of other action models. These statements give a *software requirement specification* of the Bookstore Ordering System, which means a *software requirement specification* could indeed be attained using LAMP.

5.2.8. Example output

To help the reader get an intuitive idea of our learned action models, we present the resulting models learned by our LAMP algorithm by setting the threshold δ as 0.5 and the percentage of observed intermediate states as $\frac{1}{5}$. The results of domains *briefcase*, *elevator*, *openstacks* and *trucks* are shown in Tables 17–20, where a condition in *italic* means it is needed

Table 17
The action models learned in the domain *briefcase*.

action:	move(?m - location ?l - location)
preconditions:	(is-at ?m) **(is-at ?l)**
effects:	(and *(is-at ?l)* (not (is-at ?m)) (forall (?x - portable) (when (in ?x) (at ?x ?l))) *(forall (?x - portable) (when (in ?x) (not (at ?x ?m)))))*)
action:	take-out(?x - portable)
preconditions:	(in ?x)
effects:	(and (not (in ?x)) **(forall ?l - location (when (at ?x ?l)(is-at ?l)))**)
action:	put-in(?x - portable ?l - location)
preconditions:	*(not (in ?x))* (at ?x ?l) (is-at ?l)
effects:	(and (in ?x) **(not (is-at ?l))**)

Table 18
The action models learned in the domain *elevator*.

action:	stop(?f - floor)
preconditions:	(lift-at ?f)
effects:	(and *(forall (?p - passenger) (when (and (boarded ?p) (destin ?p ?f))(not (boarded ?p))))* (forall (?p - passenger) (when (and (boarded ?p) (destin ?p ?f)) (served ?p))) (forall (?p - passenger) (when (and (origin ?p ?f) (not(served ?p))) (boarded ?p))) **(not (lift-at ?f))**)
action:	up(?f1 - floor ?f2 - floor)
preconditions:	(lift-at ?f1) (above ?f1 ?f2)
effects:	(and (lift-at ?f2) *(not (lift-at ?f1))* **(forall ?p (when (boarded ?p)(destin ?p ?f2)))**)
action:	down(?f1 - floor ?f2 - floor)
preconditions:	(lift-at ?f1) (above ?f2 ?f1)
effects:	(and (lift-at ?f2) (not (lift-at ?f1)) **(not (above ?f2 ?f1))** **(forall ?p (when (boarded ?p)(origin ?p ?f1)))**)

in the hand-written domain but is not successfully learned in our learned result, and a condition in **bold** means it is not needed in the hand-written domain but is incorrectly learned in our learned result.

In Tables 17–20, the missing conditions suggest that their corresponding weights are not high enough to be selected by LAMP, since the information constraints provided by training data are not sufficient enough to make the weights high; the additional conditions suggest that their corresponding weights are too high, such that they are selected wrongly by LAMP, since there are not enough information constraints to make the weights low. Although the results are not one hundred percent correct, they are very close to the hand-written action models or the real action models. These results can be further submitted for human editors to work on by domain analysis. For instance, in Table 17, "(is-at ?m)" and "(is-at ?l)" would not be the preconditions of "move" at the same time, since a briefcase should not be at two places at the same time according to domain analysis, which will guide us to remove one of them, and finally remove "(is-at ?l)" via further analysis of the domain. Because of the low error rates, we can see that human experts do not need to spend a lot of time on creating a new domain based on the learning result.

5.3. Discussion

In this subsection, we will summarize the major findings from our experimental results.

I In our experiments, we first validated the performance of our LAMP algorithm by varying the proportion of observed intermediate states. From Fig. 1, we can see that our algorithm performance does not vary significantly when we change our observed state numbers from 100% to only 20%. In general, the more intermediate states we observe, the better performance we have. When we validate this experiment from a state perspective, *i.e.*, we only observe a proportion of propositions, we get similar experiment results. We also vary the percent of propositions true in each state, and obtained similar results.

Table 19
The action models learned in the domain *openstacks*.

action:	setup-machine(?p - product ?avail - count)
preconditions:	(machine-available)(stacks-avail ?avail))*(not (made ?p))* **(forall (?o - order)(imply (includes ?o ?p) (started ?o)))**
effects:	(and (not (machine-available)) (machine-configured ?p))
action:	make-product(?p - product ?avail - count)
preconditions:	(machine-configured ?p)(stacks-avail ?avail) (forall (?o - order)(imply (includes ?o ?p)(started ?o)))
effects:	(and *(machine-available)*(not (machine-configured ?p)) (made ?p))
action:	start-order(?o - order ?avail ?new-avail - count)
preconditions:	(waiting ?o)(stacks-avail ?avail) *(next-count ?new-avail ?avail)* **(forall (?p - product) (imply (includes ?o ?p) (made ?p)))**
effects:	(and (started ?o)(stacks-avail ?new-avail) *(not (waiting ?o))*(not (stacks-avail ?avail)))
action:	ship-order(?o - order ?avail ?new-avail - count)
preconditions:	*(started ?o)*(stacks-avail ?avail)(next-count ?avail ?new-avail) (forall (?p - product)(imply (includes ?o ?p) (made ?p)))
effects:	(and (shipped ?o) (stacks-avail ?new-avail)(not (started ?o)) **(not (next-count ?avail ?new-avail))** (not (stacks-avail ?avail)))
action:	open-new-stack(?open ?new-open - count)
preconditions:	(stacks-avail ?open)(next-count ?open ?new-open)) **(machine-available)**
effects:	(and (stacks-avail ?new-open) *(not (stacks-avail ?open))* **(not (next-count ?open ?new-open)**))

Table 20
The action models learned in the domain *trucks*.

action:	load(?p - package ?t - truck ?a1 - truckarea ?l - location)
preconditions:	(at ?t ?l) (at ?p ?l) *(free ?a1 ?t)* (forall (?a2 - truckarea) (imply (closer ?a2 ?a1) (free ?a2 ?t)))
effects:	(and (in ?p ?t ?a1)(not (at ?p ?l))*(not (free ?a1 ?t))*)
action:	unload(?p - package ?t - truck ?a1 - truckarea ?l - location)
preconditions:	(at ?t ?l) (in ?p ?t ?a1) **(free ?a1 ?t)** (forall (?a2 - truckarea)(imply (closer ?a2 ?a1) (free ?a2 ?t)))
effects:	(and (not (in ?p ?t ?a1)) *(free ?a1 ?t)* (at ?p ?l))
action:	drive(?t - truck ?from ?to - location ?t1 ?t2 - time)
preconditions:	(at ?t ?from) (connected ?from ?to) *(time-now ?t1)* (next ?t1 ?t2)
effects:	(and (time-now ?t2) (at ?t ?to) **(not (next ?t1 ?t2))** (not (at ?t ?from)) *(not (time-now ?t1))*)
action:	deliver(?p - package ?l - location ?t1 ?t2 - time)
preconditions:	(at ?p ?l) (time-now ?t1) (le ?t1 ?t2)**(next ?t1 ?t2)**
effects:	(and (delivered ?p ?l ?t2)**(not (next ?t1 ?t2))** (at-destination ?p ?l)(not (at ?p ?l)))

II When we vary the parameter values for candidate selection, the performance varies. Choosing a good threshold value is important to the overall performance of LAMP, and we can see in general, a threshold value that is too large or too small can lead to significant reduction in performance (refer to Fig. 1 and Fig. 2). As an empirical result, our experiments show that the best threshold value is 0.5.

III From Figs. 1 and 2, we notice that some experimental domains, such as *elevator*, are more difficult to learn than other domains. This may be because domains such as *elevator* have actions that contain more conditional effects than others, and these effects are more difficult to learn.

IV In our third experiment, we validate that the efficiency of LAMP decreases when we have more plan traces (refer to Fig. 3). We also record the running time we use when different numbers of plan traces are given as input. This running time information is used to fit a curve which empirically shows that LAMP has a polynomial time complexity (refer to Fig. 5) with respect to the number of plan traces.

V We also show our LAMP algorithm can help reduce the human efforts on creating action models, which can be seen from Fig. 6. Furthermore, we give an example application of our LAMP algorithm (refer to Example 8) to show that LAMP can be applied to real-world domains, such as *software requirement engineering*.

6. Conclusions and future work

In this paper, we have presented a novel approach to learn action models with quantifiers as well as logical implications, from a set of observed plan traces where we can support partially observable intermediate states. Our LAMP learning algorithm makes use of Markov Logic Networks to learn action models automatically. Our empirical tests in four planning domains show that our LAMP algorithm is rather effective.

We list several possible directions which we could follow for our future work. Our current LAMP algorithm will enumerate all the possible preconditions and effects (or candidate formulas) according to our specific correctness constraints. When there are many actions and predicates in a planning domain, the candidate formulas will be very many, which can decrease the efficiency of LAMP. In the future, we will consider how to incorporate more domain knowledge to filter out some "impossible" candidate formulas beforehand to make the algorithm much more efficient. Domain analysis can also help reduce the errors by providing a priori what conditions are important from a domain expert's point of view. Another direction is to improve the quality of weight learning in a MLN. Currently we adopt a generative learning approach in LAMP, where we maximize the weighted pseudo log-likelihood. Other weight learning approaches such as discriminative learning might have some additional advantages. A third direction is to extend action model learning to learn elaborate action representations, including resources and functions. Finally, we will consider plan traces that contain false observations on actions and states and ways to filter out noise from the training data.

Acknowledgements

We thank the support of RGC/NSFC project N_HKUST624/09 for this research. We are grateful for the helpful comments from the editors and reviewers of the journal.

References

[1] Eyal Amir, Learning partially observable deterministic action models, in: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005), 2005, pp. 1433–1439.

[2] Scott Benson, Inductive learning of reactive action models, in: Proceedings of the Twelfth International Conference on Machine Learning (ICML 1995), 1995, pp. 47–54.

[3] Julian Besag, Statistical analysis of non-lattice data, The Statistician 24 (1975) 179–195.

[4] Jim Blythe, Jihie Kim, Surya Ramachandran, Yolanda Gil, An integrated environment for knowledge acquisition, in: Proceedings of the Sixth International Conference on Intelligent User Interfaces (IUI 2001), 2001, pp. 13–20.

[5] Brian Borchers, Judith Furman, A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems, Journal of Combinatorial Optimization 2 (4) (1998) 299–306.

[6] Lonnie Chrisman, Abstract probabilistic modeling of action, in: Proceedings of the First International Conference on Artificial Intelligence Planning Systems (AIPS 1992), 1992, pp. 28–36.

[7] Pedro Domingos Mining, Social networks for viral marketing, IEEE Intelligent Systems 20 (1) (2005) 80–82.

[8] Pedro Domingos, Toward knowledge-rich data mining, Data Mining and Knowledge Discovery 15 (2007) 21–28.

[9] Pedro Domingos, Stanley Kok, Hoifung Poon, Matthew Richardson, Parag Singla, Unifying logical and statistical AI, in: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006), 2006, pp. 2–7.

[10] Richard Fikes, Nils J. Nilsson, STRIPS: a new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (3/4) (1971) 189–208.

[11] Maria Fox, Derek Long, PDDL2.1: an extension to PDDL for expressing temporal planning domains, Journal of Artificial Intelligence Research (JAIR) 20 (2003) 61–124.

[12] Krzysztof Z. Gajos, Daniel S. Weld, Jacob O. Wobbrock, Decision-theoretic user interface generation, in: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008), 2008, pp. 1532–1536.

[13] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins, PDDL—the planning domain definition language, http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/GdKI/WS0203/pddl.pdf, 1998.

[14] Malik Ghallab, Dana Nau, Paolo Traverso, Automated Planning: Theory and Practice, Morgan Kaufmann, 2004.

[15] Yolanda Gil, Learning by experimentation: incremental refinement of incomplete planning domains, in: Proceedings of the Eleventh International Conference on Machine Learning (ICML 1994), 1994, pp. 87–95.

[16] Thomas Hernandez, Subbarao Kambhampati, Integration of biological sources: current systems and challenges ahead, SIGMOD Record 33 (3) (2004) 51–60.

[17] Jorg Hoffmann, Piergiorgio Bertoli, Marco Pistore, Web service composition as planning revisited: in between background theories and initial state uncertainty, in: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007), 2007, pp. 1013–1018.

[18] Michael P. Holmes, Charles Lee Isbell Jr., Schema learning: experience-based construction of predictive action models, in: Advances in Neural Information Processing Systems, vol. 17 (NIPS 2004), 2004.
[19] Derek Hao Hu, Qiang Yang, CIGAR: concurrent and interleaving goal and activity recognition, in: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008), 2008, pp. 1363–1368.
[20] Tuyen N. Huynh, Raymond J. Mooney, Discriminative structure and parameter learning for Markov logic networks, in: Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML 2008), 2008, pp. 416–423.
[21] Stanley Kok, Pedro Domingos, Learning the structure of Markov logic networks, in: Proceedings of the Twenty-Second International Conference on Machine Learning (ICML 2005), 2005, pp. 441–448.
[22] Stanley Kok, Pedro Domingos, Extracting semantic networks from text via relational clustering, in: Proceedings of the Nineteenth European Conference on Machine Learning (ECML 2008), 2008, pp. 624–639.
[23] Stanley Kok, Parag Singla, Matthew Richardson, Pedro Domingos, The Alchemy System for Statistical Relational AI, University of Washington, Seattle, 2005.
[24] Ugur Kuter, Evren Sirin, Bijan Parsia, Dana Nau, James Hendler, Information gathering during planning for Web Service composition, Journal of Web Semantics (JWS) 3 (2–3) (2005) 183–205.
[25] Hector J. Levesque, Fiora Pirri, Raymond Reiter, Foundations for the situation calculus, Electronic Transactions on Artificial Intelligence 2 (1998) 159–178.
[26] Dong C. Liu, Jorge Nocedal, On the limited memory BFGS method for large scale optimization, Mathematical Programming 45 (1989) 503–528.
[27] Daniel Lowd, Pedro Domingos, Efficient weight learning for Markov logic networks, in: Proceedings of the Eleventh European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2007), 2007, pp. 200–211.
[28] Lilyana Mihalkova, Raymond J. Mooney, Bottom-up learning of Markov logic network structure, in: Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML 2007), 2007, pp. 625–632.
[29] Stephen Muggleton, Luc De Raedt, Inductive logic programming: theory and methods, Journal of Logic Programming 19/20 (1994) 629–679.
[30] Megan Nance, Adam Vogel, Eyal Amir, Reasoning about partially observed actions, in: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006), 2006, pp. 888–893.
[31] Tim Oates, Paul R. Cohen, Searching for planning operators with context-dependent and probabilistic effects, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996), 1996, pp. 865–868.
[32] Hanna M. Pasula, Luke S. Zettlemoyer, Leslie Pack Kaelbling, Learning probabilistic relational planning rules, in: Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), 2004, pp. 73–82.
[33] Hanna M. Pasula, Luke S. Zettlemoyer, Leslie Pack Kaelbling, Learning symbolic models of stochastic domains, Journal of Artificial Intelligence Research 29 (2007) 309–352.
[34] Hoifung Poon, Pedro Domingos, Joint inference in information extraction, in: Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI 2007), 2007, pp. 913–918.
[35] Matthew Richardson, Pedro Domingos, Markov logic networks, Machine Learning 62 (1–2) (2006) 107–136.
[36] Gunther Sablon, Maurice Bruynooghe, Using the event calculus to integrate planning and learning in an intelligent autonomous agent, in: Current Trends in AI Planning, 1994, pp. 254–265.
[37] Matthew D. Schmill, Tim Oates, Paul R. Cohen, Learning planning operators in real-world, partially observable environments, in: Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS 2000), 2000, pp. 246–253.
[38] Shahaf Dafna, Eyal Amir, Learning partially observable action schemas, in: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006), 2006, pp. 913–919.
[39] Shahaf Dafna, Allen Chang, Eyal Amir, Learning partially observable action models: efficient algorithms, in: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006), 2006, pp. 920–926.
[40] Parag Singla, Pedro Domingos, Entity resolution with Markov logic, in: Proceedings of the Sixth IEEE International Conference on Data Mining (ICDM 2006), 2006, pp. 572–582.
[41] Parag Singla, Pedro Domingos, Markov logic in infinite domains, in: Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence (UAI 2007), 2007, pp. 368–375.
[42] Thomas J. Walsh, Michael L. Littman, Efficient learning of action schemas and web-service descriptions, in: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008), 2008, pp. 714–719.
[43] Jue Wang, Pedro Domingos, Hybrid Markov logic networks, in: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008), 2008, pp. 1106–1111.
[44] Stephen Cresswell, Thomas Leo McCluskey, Margaret Mary West, Acquisition of object-centred domain models from planning examples, in: Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009), 2009.
[45] Ron M. Simpson, Diane E. Kitchin, T.L. McCluskey, Planning domain definition using GIPO, Knowledge Engineering Review 22 (2) (2007) 117–134.
[46] T.L. McCluskey, Donghong Liu, Ron M. Simpson, GIPO II: HTN planning in a tool-supported knowledge engineering environment, in: Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), 2003, pp. 92–101.
[47] Elly Winner, Manuela Veloso, Analyzing plans with condition effects, in: Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS 2002), 2002.
[48] Tessa Lau, Pedro Domingos, Daniel S. Weld, Version space algebra and its application to programming by demonstration, in: Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Morgan Kaufmann, San Francisco, CA, 2000, pp. 527–534.
[49] R.M. Simpson, T.L. McCluskey, W. Zhao, R.S. Aylett, C. Doniat, GIPO: an integrated graphical tool to support knowledge engineering in AI planning, in: Proceedings of the European Conference on Planning, Toledo, Spain, September 2001.
[50] Xuemei 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.
[51] Qiang Yang, Wu Kangheng, Yunfei Jiang, Learning action models from plan examples with incomplete knowledge, in: Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), 2005, pp. 241–250.
[52] Qiang Yang, Wu Kangheng, Yunfei Jiang, Learning action models from plan examples using weighted MAX-SAT, Artificial Intelligence 171 (2–3) (2007) 107–143.
[53] Zhuo Hankui, Qiang Yang, Lei Li, Transfer learning action models by measuring the similarity of different domains, in: Proceedings of the Thirteenth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2009), 2009, pp. 697–704.
[54] Zhuo Hankui, Qiang Yang, Lei Li, Transferring knowledge from another domain for learning action models, in: Proceedings of the Tenth Pacific Rim International Conference on Artificial Intelligence (PRICAI 2008), 2008, pp. 1110–1115.
[55] Zhuo Hankz Hankui, Derek Hao Hu, Chad Hogg, Qiang Yang, Hector Munoz-Avila, Learning HTN method preconditions and action models from partial observations, in: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2009), 2009, pp. 1804–1810.
[56] Qiang Yang, Activity recognition: linking low-level sensors to high-level intelligence, in: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2009), 2009, pp. 20–25.

[57] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, Yves Lafon, SOAP Version 1.2, http://www.w3.org/TR/soap12-part1/.
[58] Zhuo Hankui, Lei Li, Qiang Yang, Rui Bian, Learning action models with quantified conditional effects for software requirement specification, in: Proceedings of the Fourth International Conference on Intelligent Computing (ICIC 2008), 2008, pp. 874–881.
[59] Dana Nau, Au Tsz-Chiu, Okhtay Ilghami, Ugur Kuter, Hector Munoz-Avila, J. William Murdock, Dan Wu, Fusun Yaman, Applications of SHOP and SHOP2, IEEE Intelligent Systems (2005) 34–41.
[60] Jie Yin, Xiaoyong Chai, Qiang Yang, High-level goal recognition in a wireless LAN, in: Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI 2004), 2004, pp. 578–584.