# Refining Incomplete Planning Domain Models Through Plan Traces

**Hankz Hankui Zhuo**[a], **Tuan Nguyen**[b], **and Subbarao Kambhampati**[b]
[a]Dept. of Computer Science, Sun Yat-sen University, Guangzhou, China
zhuohank@mail.sysu.edu.cn
[b]Dept. of Computer Science and Engineering, Arizona State University, US
{natuan,rao}@asu.edu

## Abstract

Most existing work on learning planning models assumes that the entire model needs to be learned from scratch. A more realistic situation is that the planning agent has an incomplete model which it needs to refine through learning. In this paper we propose and evaluate a method for doing this. Our method takes as input an incomplete model (with missing preconditions and effects in the actions), as well as a set of plan traces that are known to be correct. It outputs a "refined" model that not only captures additional precondition/effect knowledge about the given actions, but also "macro actions". We use a MAX-SAT framework for learning, where the constraints are derived from the executability of the given plan traces, as well as the preconditions/effects of the given incomplete model. Unlike traditional macro-action learners which use macros to increase the efficiency of planning (in the context of a complete model), our motivation for learning macros is to increase the accuracy (robustness) of the plans generated with the refined model. We demonstrate the effectiveness of our approach through a systematic empirical evaluation.

## 1 Introduction

Most work in planning assumes that complete domain models are given as input in order to synthesize plans. However, there is increasing awareness that building domain models at any level of completeness presents steep challenges for domain creators. Indeed, recent work in web-service composition (c.f. [Bertoli *et al.*, 2010; Hoffmann *et al.*, 2007]) and work-flow management (c.f. [Blythe *et al.*, 2004]) suggest that dependence on complete models can well be the real bottle-neck inhibiting applications of current planning technology.

Incomplete models present two challenges: how to produce robust plans despite partial knowledge [Kambhampati, 2007; Nguyen *et al.*, 2011; Weber and Bryce, 2011; Garland and Lesh, 2002] and how to improve the models over time through learning. It is this later challenge–refining incomplete models, that is the focus of the current paper. Although there has been some work on learning planning models from plan traces (c.f. [Yang *et al.*, 2007; Zhuo *et al.*, 2010; Zettlemoyer *et al.*, 2005]), most of it is aimed at learning planning models from scratch. In contrast, we are interested in the problem of refining an incomplete model through learning. In this paper we propose and evaluate a novel method for doing this. Our method takes as input a partially specified domain model (with missing preconditions and effects in the actions), as well as a set of plan traces that are known to be correct. It outputs a "refined" model that not only captures additional precondition/effect knowlege about the given actions, but also "macro actions." In the first phase, we mine candidate macros mined from the plan traces, and in the second phase we learn precondition/effect models both for the primitive actions and the macro actions. Finally we use the refined model to do planning (where the planner is biased towards using the learned macro actions where possible).

We use the MAX-SAT framework for learning, where the constraints are derived from the executability of the given plan traces, as well as the preconditions/effects of the given incomplete model [Yang *et al.*, 2007; Zhuo *et al.*, 2010]. Unlike traditional macro-action learners which use macros to increase the efficiency of planning (in the context of a complete model), our motivation for learning macros is to increase the accuracy (robustness) of the plans generated with the refined model. We demonstrate the effectiveness of our approach, called `RIM` which stands for **R**efining **I**ncomplete planning domain **M**odels through plan traces, and present a systematic empirical evaluation that demonstrates how `RIM` exploits the incomplete models as well as learned macro actions.

We organize the paper as follows. We first review related work, and then present the formal details of our framework. After that, we give a detailed description of `RIM` algorithm. Finally, we evaluate `RIM` in three planning domains and conclude our work with a discussion on future work.

## 2 Related Work

As we mentioned, there has been prior work on action model learning [Yang *et al.*, 2007; Zhuo *et al.*, 2010; Zettlemoyer *et al.*, 2005], but much of it focuses on from-scratch learning. We focus on learning in the presence of an existing incomplete model. Starting from STRIPS [Fikes *et al.*, 1972], macro-operator learning has been a staple in automated planning (c.f. [Korf, 1985; Iba, 1989; Coles and Smith, 2007; Botea *et al.*, 2005; Newton *et al.*, 2007]). All these efforts however assume that the learner has access to a complete domain model, and are motivated mainly by the desire to re-

duce the time taken for planning. In contrast, we cull and use macro-operators to increase the accuracy of the incomplete model. In this sense, our work is similar in spirit to the original case-based planning systems such as CHEF [Hammond, 1989], and PLEXUS [Alterman, 1986] that try to use plan traces ("cases") to make-up for the lack of complete domain models. The main difference is that while case-based planners focussed on case-level transfer, our approach takes a more global view of first refining the incomplete model using the learned models of primitive and macro actions.

## 3 Preliminaries and Problem Definition

A complete STRIPS domain is defined as a tuple $\mathcal{M} = \langle \mathcal{R}, \mathcal{A} \rangle$, where $\mathcal{R}$ is a set of predicates with typed objects and $\mathcal{A}$ is a set of action models. Each action model is a quadruple $\langle a, \text{PRE}(a), \text{ADD}(a), \text{DEL}(a) \rangle$, where $a$ is an action name with zero or more parameters, $\text{PRE}(a)$ is a precondition list specifying the conditions under which $a$ can be applied, $\text{ADD}(a)$ is an adding list and $\text{DEL}(a)$ is a deleting list indicating the effects of $a$. We denote $\mathcal{R}_O$ as the set of propositions instantiated from $\mathcal{R}$ with respect to a set of typed objects $O$. Given $\mathcal{M}$ and $O$, we define a planning problem as $\mathcal{P} = \langle O, s_0, g \rangle$, where $s_0 \subseteq \mathcal{R}_O$ is an initial state, $g \subseteq \mathcal{R}_O$ are goal propositions. A solution plan to $\mathcal{P}$ with respect to model $\mathcal{M}$ is a sequence of actions $p = \langle a_1, a_2, \ldots, a_n \rangle$ that achieves goal $g$ starting from $s_0$.

An action model $\langle a, \text{PRE}(a), \text{ADD}(a), \text{DEL}(a) \rangle$ is considered *incomplete* if there are predicates missing in $\text{PRE}(a)$, $\text{ADD}(a)$, or $\text{DEL}(a)$. We denote $\widetilde{\mathcal{A}}$ as a set of incomplete action models, $\widetilde{\mathcal{R}}$ the set of predicates used in the specification of $\widetilde{\mathcal{A}}$, and thus $\widetilde{\mathcal{M}} = \langle \widetilde{\mathcal{R}}, \widetilde{\mathcal{A}} \rangle$ the corresponding incomplete STRIPS domain. Note that although action models in $\widetilde{\mathcal{A}}$ might have incomplete preconditions and effects, we assume that no action model in $\mathcal{A}$ is missing, and that preconditions and effects specified in $\widetilde{\mathcal{A}}$ are correct.

We denote $\mathcal{T}$ as a set of plan traces, where each provides a successful plan $p_i$ for problem $\mathcal{P}_i = \langle O^i, s_0^i, g^i \rangle$. A macro-operator, a set of which is denoted by $\mathcal{O}$, is defined by a tuple $\langle o, \alpha(o), \text{PRE}(o), \text{ADD}(o), \text{DEL}(o) \rangle$, where $o$ is its name with zero or more parameters, $\alpha(o)$ is an action sequence that constitutes $o$, $\text{PRE}(o)$, $\text{ADD}(o)$, and $\text{DEL}(o)$ respectively are the precondition, add and delete lists of $o$. A tuple $\langle o, \alpha(o) \rangle$ is called a *macro-operator schema*.

In this work, we are given a triple $\langle \widetilde{\mathcal{P}}, \widetilde{\mathcal{M}}, \mathcal{T} \rangle$ of planning problem $\widetilde{\mathcal{P}} = \langle O, s_o, g \rangle$, incomplete domain $\widetilde{\mathcal{M}}$ and set of successful plan traces $\mathcal{T}$. Our algorithm first refines the domain $\widetilde{\mathcal{M}}$ with more complete action models and new macro-operators, resulting in the refined domain $\widetilde{\mathcal{M}}_R$, and then use it to solve the problem $\widetilde{\mathcal{P}}$. We note that the model refinement needs to be done only once, and the refined domain model $\widetilde{\mathcal{M}}_R$ could be used to solve new planning problems.

An example input of our planning problem in *blocks*[1] is shown in Figure 1, which is composed of three parts: incomplete action models (Figure 1(a)), initial state $s_0$ and goal $g$ (Figure 1(b)), and a plan example set (Figure 1(c)). In Figure 1(a), the dark parts indicate the missing predicates. In Figure 1(c), $p_1$ and $p_2$ are two plan traces, where initial states and goals are bracketed. An example output is a solution to the planning problem given in Figure 1, i.e., "`(unstack C A)(putdown C)(pickup B)(stack B A)(pickup C)(stack C B)(pickup D)(stack D C)`".

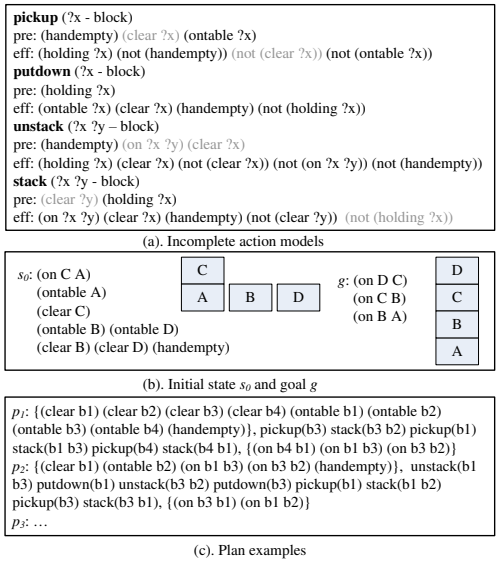


Figure 1: An input example of `RIM` for the *blocks* domain

## 4 The `RIM` Approach

Our `RIM` approach consists of two phases. In the first phase, we learn macro-operators and action models with the given plan traces $\mathcal{T}$ and the incomplete domain $\widetilde{\mathcal{M}}$. In the second phase, we exploit the learned macro-operators and action models in solving new planning problems. In the subsequent subsections, we first address the learning phase in detail, and then describe the planning phase briefly.

Our learning framework (the first phase) can be found in Algorithm 1. It begins with the step of collecting sets of predicates $P$, action schemas $A$ and macro-operator schemas $O$ from incomplete action models $\widetilde{\mathcal{A}}$ and plan traces $\mathcal{T}$. In the next two steps, it constructs sets of soft and hard constraints to ensure that the learned domain model can best explain the input plan traces and incomplete action models. Finally, we solve these constraints using a weighted MAX-SAT solver to obtain sets of macro-operators and (refined) action models.

**Algorithm 1** Phase I: refining domain models

**Input:** incomplete action models $\widetilde{\mathcal{A}}$, and plan traces $\mathcal{T}$.
**Output:** macro-operators $\mathcal{O}$ and action models $\mathcal{A}$.

1: build sets of predicates, action schemas and macro-operator schemas: $(P, A, \mathcal{O})$=*build_schemas*($\widetilde{\mathcal{A}}$, $\mathcal{T}$);
2: build soft constraints: *state constraints*, *pair constraints*;
3: build hard constraints: *macro constraints*, *action constraints*, *plan constraints* and *incompleteness constraints*;
4: solve all constraints, and build $\mathcal{O}$ and $\mathcal{A}$;
5: **return** $\mathcal{O}$ and $\mathcal{A}$;

[1] http://www.cs.toronto.edu/aips2000/

### 4.1 Generating Predicates, Action and Macro-operator Schemas

It is straightforward to construct the set of predicates $P$. We first collect all predicates $\widetilde{\mathcal{R}}$ used in the given incomplete action models $\widetilde{\mathcal{A}}$, and view them as the initial set of predicates $P$. We then scan each proposition in plan traces and put its corresponding predicates (by replacing parameters of the proposition with variables) in $P$ if it is not in $P$. Likewise, we build a set of action schemas $A$ by scanning all the incomplete action models and plan traces.

There are no easy ways to construct macro-operator schemas, since they neither exist in the plan traces nor in the incomplete action models. We propose to construct macro-operator schemas with the help of frequent pattern mining techniques and incomplete action models. In particular, we consider an action subsequence that satisfies the following two conditions to be a macro-operator schema: (1) it frequently occurs in plan traces, the insight of which implicitly suggests they are more likely to be used in future problem solving; (2) its actions have strong connections with each other with respect to incomplete action models, which suggests these actions are more likely to be successfully executed (i.e., with respect to the complete domain model). Note that we define the strength $\theta(o)$ of a macro-operator $o$ as $\theta(o) = \frac{\#supported\_preconditions}{\#preconditions}$, where $\#supported\_preconditions$ is the number of preconditions supported by some actions in $o$, and $\#preconditions$ is the total number of preconditions of actions in $o$.

Any action subsequence can be considered a macro-operator schema. However, learning too many macro-operators is costly. We thus choose to eliminate those with low frequency or strength by setting a support constant $\delta_0$ for frequency and a threshold $\theta_0$ for strength.We borrow the notion of frequent patterns defined in [Zaki, 2001; Pei *et al.*, 2004] to mine the frequent plan fragments. The problem of mining sequential patterns can be stated as follows. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a set of $n$ items. We call a subset $X \subseteq \mathcal{I}$ an itemset and $|X|$ the size of $X$. A sequence is an ordered list of itemsets, denoted by $s = \langle s_1, s_2, \ldots, s_m \rangle$, where $s_j$ is an itemset. The size of a sequence is the number of itemsets in the sequence, i.e., $|s| = m$. The length $l$ of a sequence $s = \langle s_1, s_2, \ldots, s_m \rangle$ is defined as $l = \sum_{i=1}^{m} |s_i|$. A sequence $s_a = \langle a_1, a_2, \ldots, a_n \rangle$ is a *subsequence* of another sequence $s_b = \langle b_1, b_2, \ldots, b_m \rangle$ if there exist integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \ldots, a_n \subseteq b_{i_n}$, denoted by $s_a \sqsubseteq s_b$. A sequence database $S$ is a set of tuples $\langle sid, s \rangle$, where $sid$ is a sequence $id$ and $s$ is a sequence. A tuple $\langle sid, s \rangle$ is said to *contain* a sequence $a$, if $a$ is a subsequence of $s$. The support of a sequence $a$ in a sequence database $S$ is the number of tuples in the database containing $a$, i.e., $sup_S(a) = |\{\langle sid, s \rangle | (\langle sid, s \rangle \in S) \cap (a \sqsubseteq s)\}|$. Given a positive integer $\delta$ as the support threshold, we call $a$ a *frequent* sequence if $sup_S(a) \geq \delta$. Given a sequence database and the support threshold, frequent sequential pattern mining problem is to find the complete set of sequential patterns whose supports are larger than the threshold.

We convert the set of plan traces to a sequence database in order to make use of the frequent pattern mining algorithm for extracting macro-operators. Given that different action instances with the same action name share the same model description (i.e., preconditions/effects), we view each action name (parameters omitted) in plan traces as an itemset, which has only one element, and a plan trace as a sequence. The set of plan traces can now be viewed as a sequence database. In addition, we restrict the indices of itemsets of the mined frequent subsequence to be continuous. In this way, we can exploit a frequent pattern mining algorithm, such as SPADE [Zaki, 2001], to mine a set of frequent subsequences.

Furthermore, after mining a set of frequent action subsequences, we consider the parameter constraints of actions. For example, consider frequent action subsequence "`putdown unstack stack` ". There may be two scenarios in two different plan traces, which are "`(putdown A)(unstack B C)(stack B A)` " and "`(putdown a)(unstack b c)(stack b d)` ". These two subsequences should not be seen as the same macro-operator, since they represent different meanings. Specifically, the first scenario produces an effect "`(on B A)` ", while the second scenario produces an effect "`(clear a)` ". Note that objects "A" and "a" are two instances of the same variable, likewise for other objects. As such, we consider these two scenarios as two macro-operator schemas, as shown in Table 1.

Table 1: The example macro-operator schemas

| |
|---|
| (:macro macro1<br>(:parameters ?x - block ?y - block ?z - block)<br>(:actions (putdown ?x) (unstack ?y ?z) (stack ?y ?x))) |
| (:macro macro2<br>(:parameters ?x - block ?y - block ?z - block ?w - block)<br>(:actions (putdown ?x) (unstack ?y ?z) (stack ?y ?w))) |

To sum up, we perform the following three steps to generate macro-operator schemas:

1. We first mine a set of subsequences $\mathcal{F}$ from $\mathcal{T}$, whose frequencies are larger than the preset support constant $\delta_0$, neglecting the parameters of actions in $\mathcal{T}$.
2. We then take the parameters of actions in $\mathcal{F}$ into consideration, obtaining a new set of action subsequences with corresponding parameters. We eliminate parameterized subsequences whose frequencies are smaller than $\delta_0$ and whose strengths are larger than the preset constant $\theta_0$, resulting in a new set of frequent subsequences $\mathcal{F}'$.
3. Finally, we build macro-operator schemas $\mathcal{O}$ from action subsequences in $\mathcal{F}'$ with all corresponding parameters. As mentioned above, Table 1 shows example macro-operator schemas constructed from action subsequences "`(putdown ?x)(unstack ?y ?z)(stack ?y ?x)` " and "`(putdown ?x)(unstack ?y ?z)(stack ?y ?w)` ".

### 4.2 Building Soft Constraints

The next step in `RIM` enforces several constraints on all possible complete precondition and effect descriptions of actions and macro-operators using the inputted incomplete action models. These constraints are designed to be soft, directing the MAX-SAT algorithm towards learning the most probable complete description of actions and macro-operators.

**State Constraints**

We first build soft constraints encoding possible preconditions and effects of actions and macro-operators implied by state transitions in plan traces. We preprocess plan traces using incomplete action models to obtain more state information for building state constraints. To do this, we simply "execute" each plan trace starting from its initial state, and calculate (incomplete) states between actions using incomplete action models. In particular, given a plan trace $t = \langle s_0, a_1, \ldots, s_{n-1}, a_n, g \rangle$, we execute $t$ from $s_0$ using the incomplete action models $\widetilde{\mathcal{A}}$, and calculate states $s_i$ as follows: $s_i = \widetilde{\text{ADD}}(a_i) \cup \widetilde{\text{PRE}}(a_{i+1}) \cup \widetilde{\text{DEL}}(a_{i+1})$.

Note that in defining $s_i$ we do not consider information from previous states $s_j$ ($j < i$) due to the incompleteness of models (i.e., we cannot determine whether propositions in previous states are deleted by their next actions when propositions are not in the delete lists of these actions).[2] We also assume that actions do not delete propositions that are nonexistent, i.e., if $p \in \widetilde{\text{DEL}}(a_{i+1})$, $p$ should be in $a_{i+1}$'s previous state $s_i$. We denote the resulting set of plan traces by $\mathcal{T}'$.

By observation, in $\mathcal{T}'$ we find that if a predicate frequently appears before an action or macro-operator is executed, and its parameters are also parameters of the action or operator, then the predicate is likely to be its precondition. Similarly, if a predicate frequently appears after an action or operator is executed, it is likely to be one of its effects. We encode this information in the form of *state constraints* as follows:[3]

1. For each predicate $p$ in the state *where* action $a$ is executed and $\text{PARA}(p) \subseteq \text{PARA}(a)$, we have $p \in \text{PRE}(a)$.
2. For each predicate $p$ in the state *where* operator $o$ is applied and $\text{PARA}(p) \subseteq \text{PARA}(o)$, we have $p \in \text{PRE}(o)$.
3. For each predicate $p$ in the state *after* action $a$ is executed and $\text{PARA}(p) \subseteq \text{PARA}(a)$, we have $p \in \text{ADD}(a)$.
4. For each predicate $p$ in the state *after* operator $o$ is applied and $\text{PARA}(p) \subseteq \text{PARA}(o)$, we have $p \in \text{ADD}(o)$.

We denote the set of the above constraints by SC. We scan all plan traces in $\mathcal{T}'$ and count the occurrences of each constraint in $\mathcal{T}'$. We assign the number of occurrences as the weight of the corresponding constraint.

**Pair Constraints**

It is likely that actions that frequently occur together have intimate relationships, such as one action providing conditions for its next action. We would like to capture this information to help learn action models.

Let $\alpha(o) = \langle a_1, \ldots, a_n \rangle$ be an action sequence of macro-operator $o \in O$. Since $\langle a_1, \ldots, a_n \rangle$ frequently occur together, as presented in Step 1 of Algorithm 1, $a_i$ is likely providing conditions for $a_{i+1}$ ($0 < i < n$), whose parameters are included by both $a_i$ and $a_{i+1}$. We formulate the idea with the following constraints: for each predicate $p$, if $\text{PARA}(p) \subseteq (\text{PARA}(a_i) \cap \text{PARA}(a_{i+1}))$, then $p \in \text{ADD}(a_i) \wedge p \in \text{PRE}(a_{i+1})$.

We call these constraints *pair constraints*. The weights of these constraints are the frequencies the macro-operators, computed when building macro-operator schemas.

## 4.3 Building Hard Constraints

In this subsection, we enforce a set of hard constraints that must be satisfied by action models and macro-operators.

**Macro Constraints**

Given action sequence $\alpha(o) = \langle a_1, \ldots, a_n \rangle$ of $o$ and models of each action $a_i$, we require that preconditions of $o$ should provide sufficient conditions for executing all actions $a_i$; effects of $o$ should include those that are created and not deleted by the action sequence. That is to say, the following constraints should hold:

1. For each action $a_i$ and predicate $p$, if $p \in \text{PRE}(a_i)$ and there is no action $a_j$ prior to $a_i$ that adds $p$, then $p \in \text{PRE}(o)$ holds.
2. For each action $a_i$ and predicate $p$, if $p \in \text{ADD}(a_i)$ and there is no action $a_j$ after $a_i$ that deletes $p$ and $p \notin \text{PRE}(o)$, then $p \in \text{ADD}(o)$ holds.
3. For each action $a_i$ and predicate $p$, if $p \in \text{DEL}(a_i)$ and there is no action $a_j$ after $a_i$ that adds $p$ and $p \in \text{PRE}(o)$, then $p \in \text{DEL}(o)$ holds.

**Action Constraints**

To make sure that the learned action models are consistent with the STRIPS language, we further enforce some constraints, called action constraints, on different actions. We formulate the constraints as follows [Yang *et al.*, 2007] and denote them by AC:

1. An action may not add a *fact* (instantiated atom) which already exists before the action is applied. This constraint can be encoded as: $p \in \text{ADD}(a) \Rightarrow p \notin \text{PRE}(a)$.
2. An action may not delete a *fact* which does not exist before the action is applied. This constraint can be encoded as: $p \in \text{DEL}(a) \Rightarrow p \in \text{PRE}(a)$.

**Plan Constraints**

We require that the action models learned do not violate the correctness of plan traces. This requirement is imposed on the relationship between ordered actions in plan traces, ensuring 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 $p$. We formulate the constraints as follows and denote them by PC:

$$p \in \text{PRE}(a_j) \wedge (p \in s_0 \vee (p \in \text{ADD}(a_i) \wedge \neg p \notin \text{DEL}(a_k))),$$

where $i < k < j$.

**Incomplete Model Constraints**

Finally, we enforce constraints ensuring that preconditions and effects in the given action models, though incomplete, are correctly specified. In other words, for each action $a \in A$ and predicate $p$, we have

$$p \in \widetilde{\text{PRE}}(a) \rightarrow p \in \text{PRE}(a),$$

[2] An alternative approach that more fully exploits the partial model, that we hope to investigate in future, is to allow previous state information, but make the weight of the persisting conditions to be lower than the immediate conditions

[3] We denote $\text{PARA}(p)$, $\text{PARA}(a)$ and $\text{PARA}(o)$ as the set of parameters involed in predicate $p$, action $a$ and macro-operator $o$.

and

$$p \in \widetilde{\text{ADD}}(a) \rightarrow p \in \text{ADD}(a),$$

and

$$p \in \widetilde{\text{DEL}}(a) \rightarrow p \in \text{DEL}(a).$$

To make sure these constraints are hard, we assign a large enough weight, denoted by $w_{max}$, to these constraints. In our experiment, we simply chose the maximal weight of state constraints and macro constraints as the value of $w_{max}$.

### 4.4 Solving Constraints

We put all constraints together and solve them with a weighted MAX-SAT solver [LI *et al.*, 2007]. We exploit MaxSatz [LI *et al.*, 2007] to solve all the hard constraints, and attain a *true* or *false* assignment to maximally satisfy the weighted constraints. Given the solution assignment, the construction of macro-operators $\mathcal{O}$ and action models $\mathcal{A}$ is straightforward; e.g., if "$p \in \text{ADD}(a)$" is assigned *true* in the result of the solver, $p$ will be converted into an effect of $a$.

### 4.5 Phase II: Solving Planning Problems

With the macro-operators and action models learned, we can easily solve new planning problems using planners, such as FF.[4] We view each macro-operator as a special action model during planning, neglecting its corresponding action sequence. We make a minor modification to FF to make it prefer applying macro-operators during searching. Macros in the plan returned will then be replaced by corresponding action subsequences, resulting in the solution for problem $\tilde{\mathcal{P}}$.

## 5 Experiments

### 5.1 Dataset and Criterion

We evaluate RIM algorithm in three planning domains: *blocks*[2], *driverlog*[5] and *depots*[4]. In each domain, we generate from 30 to 150 plan traces for learning domain models and 50 new planning problems for testing the learnt domain models.

We define the accuracy $Acc$ as the percentage of correctly solved planning problems. Specifically, we employ RIM to generate solutions to planning problems, and execute these solutions from the initial states using *ground truth* action models which are assumed to be correct. In other words, we assume that we possess a set of ground truth action models for testing RIM. If a solution can be successfully executed, achieving the corresponding goals, then it is considered to be *correct*. We denote by $N_c$ the number of planning problems solved by these correct solutions, and by $N_t$ the number of all testing problems. As a result, the accuracy of RIM can be defined by $Acc = \frac{N_c}{N_t}$.

### 5.2 Experimental Results

We evaluate RIM in the following aspects. We first compare the accuracies of solving planning problems using domain models learnt by RIM and ARMS, to see the advantage of the learnt macro-operators in solving planning problems. We then test the variation of accuracies of RIM with respect to different percentages of completeness of domain models and different thresholds of frequencies forming macro-operators. Finally, we show the running time to see the efficiency of RIM. In all experiments, we set the threshold for strength of all macro-operators $\theta_0 = 0.25$.

**Comparison between RIM and ARMS**

To see the benefit we get from the learnt macro-operators, we compare RIM and ARMS like this: we first learn macro-operators and action models using RIM and solve 50 new planning problems using the learnt models; we then learn action models using ARMS and solve the same 50 planning problems with the learnt action models. Note that since ARMS constructs action models from only plan traces, we also assume empty action models in using RIM. Thus, the only difference is that RIM learns both action models and macro-operators to further support robust plan synthesis. The threshold $\delta_0$ is set to be 15.

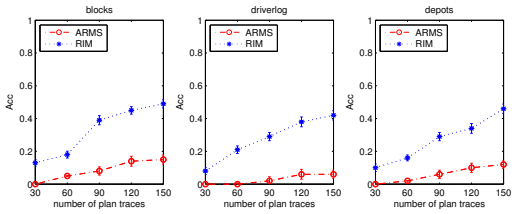


Figure 2: Comparison between ARMS and RIM

Figure 2 shows the accuracies of the two approaches. We can see that the accuracies of RIM are generally better than ARMS, which suggests that the learned macro-operators indeed help solve new planning problems. This is because macro-operators with high frequencies contribute helpful information for searching correct actions. We can also find that as expected, when the number of plan traces increases, the accuracies are also getting higher. This is consistent with our intuition, since the more plan traces we have, the more information is available for learning high-quality domain models (including both macro-operators and action models), and thus helpful for solving new planning problems. It is likely that RIM may produce longer solutions because of its preferences for using macro-operators. However, in the experiments we observe that the average length of solutions of RIM is not significantly higher, compared to not using macro-operators. For example, consider using 150 plan traces for learning in the *blocks* domain. The average length of solutions is 18 when using action models learnt by ARMS; while the average length of solutions (to the same problems as solved by ARMS) is 21 when using preferences of macro-operators learnt by RIM. This is reasonable given that the plans with macro-operators have higher quality.

**Accuracies with respect to incomplete action models**

We also would like to see the impact of the input incomplete action models. We vary the percentage of *known* preconditions or effects of action models from 20% to 100%, and run RIM three times to calculate the average for accuracies. We fix the number of plan traces to be 90 and keep the same threshold of frequencies $\delta_0 = 15$ as the previous part.

[4]http://fai.cs.uni-saarland.de/hoffmann/ff.html

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

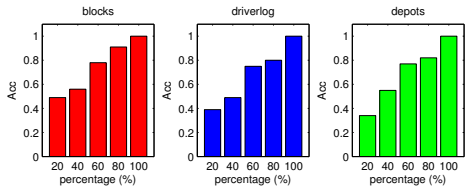


Figure 3: Accuracies w.r.t. completeness of action models.

The result is shown in Figure 3. We find that in all three domains the accuracies generally get higher when the percentages for model completeness increase. The results indicate that the input incomplete action models encoded as hard constraints in RIM are helpful for acquiring high-quality macro-operators and action models. As an extreme case, when the input action models are complete, i.e., the percentage is 100%, all test problems can be correctly solved given the learned domain models.[6] This is because the macro-operators learned are also complete based on the hard macro constraints, which suggests that we can solve the test problems correctly using the learnt macro-operators.

**Accuracies with respect to frequency thresholds**

We also studied the impact of different frequency thresholds $\delta_0$, which is used in mining macro-operators, in synthesizing correct plans. We set the number of plan traces to be 90, and the percentage of known preconditions and effects of action models to be 60%. The result is shown in Figure 4.

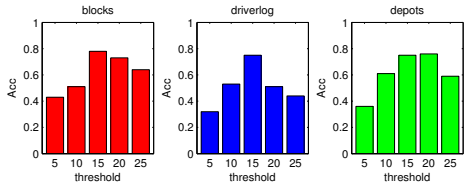


Figure 4: Accuracies w.r.t. different thresholds of frequencies.

We find that the accuracies in all three domains go up at the beginning and then go down after the threshold $\delta_0$ reaches a large value (e.g., 15). This indicates that the threshold should neither be too low nor too high, since many noisy macro-operators will be learned when the threshold is too low, and many high-quality macro-operators will be missed when it is too high. The correctness of solution plans decreases in both cases. For our experiment data, we empirically observe the best threshold should be 15 with respect to 90 plan traces.

The plots in Figure 4 also shed light on whether macros provide additional benefit when learning in conjunction with partial models. For a given partial model, when we increase the threshold for macro frequency too high–thereby effectively eliminating macros from the learned model–the planner accuracy reduces. This demonstrates that even with partial models (domain knowledge), macros do help in improving the planner accuracy. Part of the reason is that the learned macro actions improve the learned primitive actions too. This synergy happens because we build pair constraints based on macro actions as well as partial model to improve the learning of primitive actions.

[6]We use only solvable test problems in the experiment.

**Running time**

To study the efficiency of RIM, we ran RIM over 50 planning problems and calculated the average solving time with respect to different number of plan traces in the *driverlog* domain. The result is shown in Figure 5. As can be seen from the figure, the running time increases polynomially with the number of input plan traces. This can be verified by fitting the relationship between the number of plan traces and the running time to a performance curve with a polynomial of order 2 or 3. For example, the fit polynomial in Figure 5 is $-0.0289x^2 + 14.62x - 39.6$. The results for the other two domains are similar to *driverlog*, i.e., the running time also polynomially increases as plan traces increase.

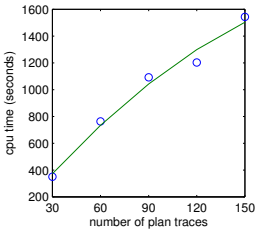


Figure 5: The running time of RIM for domain *driverlog*.

## 6 Conclusion

In this paper, we presented a system called RIM for learning domain models for planning with incomplete models. RIM is able to integrate knowledge from both incomplete domain models and a set of plan traces to produce solutions to new planning problems. With the incomplete domain models and plan traces, we first learn a set of macro-operators as well as a set of updated action models, and then solve new planning problems using the learned models. Our approach is well suited for scenarios where the planner is limited to incomplete models of the domain, but does have access to a set of plan traces that are correct with respect to the complete (but unknown) domain theory. Although in the current paper we focused on incomplete but correct initial models, our approach can also handle incorrect initial models. All that is needed is to make the precondition and effect constraints also be soft constraints rather than hard as they are in the current setup. Another optional solution to our problem is to directly mine a set of *frequent* (i.e., highly related to the problem) plan fragments and "concatenate" these fragments with the help of incomplete models to form the final solution, rather than refining incomplete domain models, as presented in our parallel work ML-CBP [Zhuo *et al.*, 2013]. In the future we are interested in comparing RIM and ML-CBP.

**Acknowledgements:** Hankz Hankui Zhuo thanks Natural Science Foundation of Guangdong Province of China (No. S2011040001869), Research Fund for the Doctoral Program of Higher Education of China (No. 20110171120054) and National Natural Science Foundation of China (61033010) for the support of this research. Kambhampati's research is supported in part by the ARO grant W911NF-13-1-0023, the ONR grants N00014-13-1-0176, N00014-09-1-0017 and N00014-07-1-1049, and the NSF grant IIS201330813.

## References


[Alterman, 1986] Richard Alterman. An adaptive planner. In *Proceedings of AAAI*, pages 65–71, 1986.

[Bertoli *et al.*, 2010] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence Journal*, 174(3-4):316–361, 2010.

[Blythe *et al.*, 2004] J. Blythe, E. Deelman, and Y. Gil. Automatically composedworkflows for grid environments. *IEEE Intelligent Systems*, 19(4):16–23, 2004.

[Botea *et al.*, 2005] Adi Botea, Markus Enzenberger, Martin Muller, and Jonathan Schaeffer. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[Coles and Smith, 2007] Andrew Coles and Amanda Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, 2007.

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Niisson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Garland and Lesh, 2002] Andrew Garland and Neal Lesh. Plan evaluation with incomplete action descriptions. In *Proceedings of AAAI*, pages 461–467, 2002.

[Hammond, 1989] K. J. Hammond. *Case-Based Planning: Viewing Planning as a Memory Task*. Academic Press, San Diego, CA, 1989.

[Hoffmann *et al.*, 2007] Joerg Hoffmann, Piergiorgio Bertoli, and Marco Pistore. Web service composition as planning, revisited: In between background theoriesandinitial state uncertainty. In *Proceedings of AAAI*, 2007.

[Iba, 1989] Glenn A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285–317, 1989.

[Kambhampati, 2007] Subbarao Kambhampati. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain theories. In *Proceedings of AAAI*, 2007.

[Korf, 1985] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[LI *et al.*, 2007] Chu Min LI, Felip Manya, and Jordi Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, October 2007.

[Newton *et al.*, 2007] M.A. Hakim Newton, John Levine, Maria Fox, and Derek Long. Learning macro-actions for arbitrary planners and domains. In *Proceedings of ICAPS*, pages 256–263, 2007.

[Nguyen *et al.*, 2011] Tuan Nguyen, Subbarao Kambhampati, and Minh Do. Synthesizing robust plans under incomplete domain models. In *AAAI Workshop on Generalized Planning*, 2011.

[Pei *et al.*, 2004] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1424–1440, 2004.

[Weber and Bryce, 2011] Christopher Weber and Daniel Bryce. Planning and acting in incomplete domains. In *Proceedings of ICAPS*, pages 274–281, 2011.

[Yang *et al.*, 2007] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence Journal*, 171:107–143, February 2007.

[Zaki, 2001] Mohammed J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *machine learning*, 42:31–60, 2001.

[Zettlemoyer *et al.*, 2005] Luke S. Zettlemoyer, Hanna M. Pasula, and Leslie Pack Kaelbling. Learning planning rules in noisy stochastic worlds. In *Proceedings of AAAI*, 2005.

[Zhuo *et al.*, 2010] Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and implications. *Artificial Intelligence*, 174(18):1540 – 1569, 2010.

[Zhuo *et al.*, 2013] Hankz Hankui Zhuo, Tuan Nguyen, and Subbarao Kambhampati. Model-lite case-based planning. In *Proceedings of AAAI*, 2013.