Artificial Intelligence 246 (2017) 1–21

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

# Model-lite planning: Case-based vs. model-based approaches

Hankz Hankui Zhuo [a,*], Subbarao Kambhampati [b]

[a] School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
[b] Department of Computer Science and Engineering, Arizona State University, United States

ARTICLE INFO

*Article history:*
Received 2 July 2015
Received in revised form 17 January 2017
Accepted 19 January 2017
Available online 25 January 2017

*Keywords:*
AI planning
Case-based planning
Action model learning

ABSTRACT

There is increasing awareness in the planning community that depending on complete models impedes the applicability of planning technology in many real world domains where the burden of specifying complete domain models is too high. In this paper, we consider the problem of generating robust and accurate plans, when the agent only has access to incomplete domain models, supplanted by a set of successful plan cases. We will develop two classes of approaches – one case-based and the other model-based. `ML-CBP` is a case-based approach that leverages the incomplete model and the plan cases to solve a new problem directly by affecting case-level transfer. `RIM` is a model-based approach that uses the incomplete model and the plan cases to first learn a more complete model. This model contains both primitive actions as well as macro-operators that are derived from the plan cases. The learned model is then used in conjunction with an off-the-shelf planner to solve new problems. We present a comprehensive evaluation of the two approaches, both to characterize their relative tradeoffs, and to quantify their advances over existing approaches.



## 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 (cf. [5,23]) and work-flow management (cf. [6]) suggest that dependence on complete models can well be the real bottle-neck inhibiting applications of current planning technology.

There has thus been interest in the so-called "model-lite" planning approaches (cf. [29]) that aim to synthesize plans even in the presence of incomplete domain models. The premise here is that while complete models cannot be guaranteed, it is often possible for the domain experts to put together reasonable but incomplete models to generate plans. The challenge then is to work with these incomplete domain models, and yet produce plans that have a high chance of success with respect to the "complete" (but unknown) domain model. Producing such robust plans is, of course, only possible if the planner has access to additional sources of knowledge besides the incomplete domain model.

In this paper, we look at the case where, in addition to the incomplete model, the agent has access to plan cases that were successful in the past. It is not difficult to collect successful plan cases in real-world applications. For example, operating systems, such as Linux, can easily collect successful plan cases by recording the log information when users operating the system, as mentioned by Yang et al. [52]. As another real-world application, in intelligent factory, the control center

* Corresponding author.
*E-mail addresses:* zhuohank@mail.sysu.edu.cn (H.H. Zhuo), rao@asu.edu (S. Kambhampati).

http://dx.doi.org/10.1016/j.artint.2017.01.004
0004-3702/

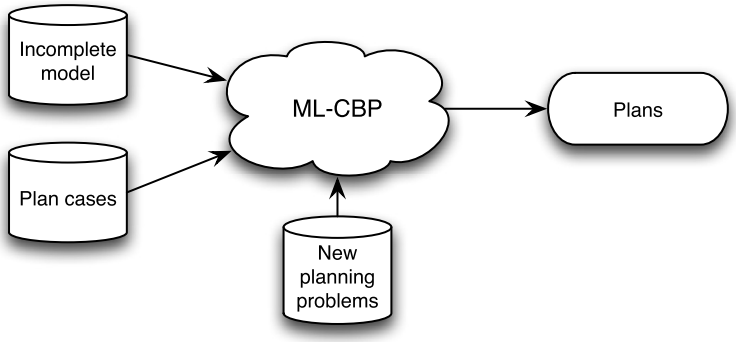


**Fig. 1.** The framework of ML-CBP.

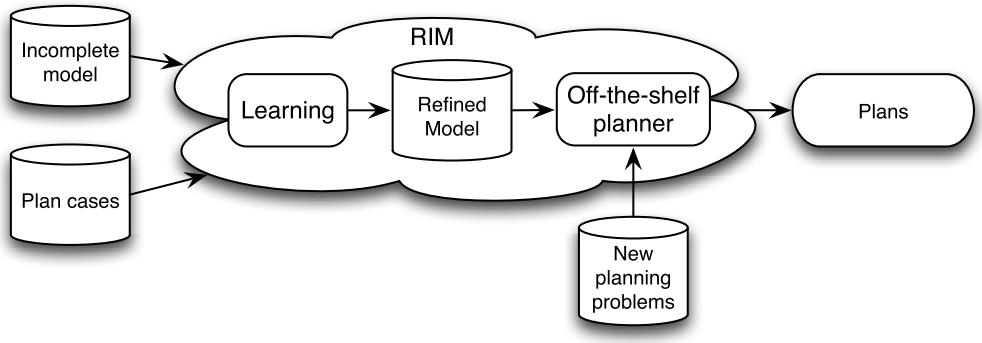


**Fig. 2.** The framework of RIM.

can collect plan cases by recording instructions distributed to mechanical equipments and/or workers, or using sensors to observe actions executed by mechanical equipments and/or workers. On the other hand, we consider that the preconditions and effects specified in the incomplete domain model are composed of human's common knowledge. For example, Executing an action of "deleting a file" generally requires a precondition specifying that "the file exists", resulting in "the file being deleted" (as an effect of the action of "deleting"). This common knowledge is generally correct and not difficult to collect. Here we define a plan case by a triple, i.e., an initial state $s_0$, a goal $g$ and an action sequence that transits $s_0$ to $g$. These cases can be seen as providing additional knowledge of the domain over and above the incomplete domain theory. Specifically, they indicate that the true model must correspond to an extension of the incomplete model that will allow the plan cases to execute successfully. Such cases can be exploited in either case-based or model-based approaches. In the case-based approaches, the new problems are solved by directly using cases, while in the later, the cases are first compiled into a (refined) action model, which is then passed on to an off-the-shelf planner. The aim of this paper is to develop and compare both case-based and model-based approaches. We are interested, in particular, in how the number of available cases and the degree of incompleteness of the input model affect the relative performance of the two types of approaches.

At first blush, it would seem as if we can directly use off-the-shelf case-based approaches (such as OAKPlan [44]) or action-model-learning approaches (such as ARMS [60,52]). This however turns out not to be the case. As we shall elaborate below, the existing case-based approaches assume availability of the complete domain model (and focus on efficiency rather than robustness). Most of existing model-learning approaches, in contrast, assume that only plan cases are available, and thus do not exploit the availability of a partial/incomplete model. As we shall see, such a partial model can be leveraged to come up with models that include macro-actions in addition to primitive ones.

Accordingly, in this paper, we develop and present novel case-based as well as model-based approaches. For the case-based approach, we take a two stage process. First, we use the incomplete domain model to synthesize a "skeletal" plan. Next, with the skeletal plan in hand, we "mine" the case library for fragments of plans that can be spliced into the skeletal plan to increase its correctness. The plan improved this way is returned as the best-guess solution to the original problem. We call this approach ML-CBP, which stands for **M**odel-**L**ite **C**ase-**B**ased **P**lanning, a previous version of which can be found from [57]. The framework of ML-CBP is shown in Fig. 1. Plan cases are exploited to produce plan solutions for planning problems directly in ML-CBP.

For the model-based approach we develop a framework called RIM (which stands for **R**efining **I**ncomplete planning domain **M**odels through plan cases, a previous version of which can be found from [58]. The framework of RIM is shown in Fig. 2) that extends the ARMS family of methods for learning models from plan cases [60,52] so that they can benefit from both the incomplete model, and the macro-operators extracted from the plan cases. In the first phase, we mine candidate macros mined from the plan cases, 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). In RIM, plan cases are leveraged to refine action models and learn macro-operators (which are then utilized to generate plan solutions), rather than being used to directly generate plan solutions for planning problems, as done by ML-CBP. We refer to RIM as *model-based approach*.

The core of our paper involves presenting and evaluating ML-CBP and RIM. The aim of our evaluation is three fold: First, and foremost, we are interested in comparing the relative advantages of model-based vs. case-based approaches for model-lite planning. We do this by comparing ML-CBP and RIM in terms of the accuracy (robustness) of the plans they produce given the same input (incomplete) model, and cases. In doing this comparison, we also evaluate how the performance varies as we vary the level of incompleteness of the input model, as well as the number of plan cases provided. Finally, in addition to accuracy, we also consider the computational resources consumed by each of these approaches. Secondly, we also compare the performance of ML-CBP and RIM to an off-the-shelf case-based planning system (OAKPlan [44]) and an action-model learning framework (ARMS [60,52]). This comparison serves to establish the technical advances of our proposed methods over existing work.

We organize the paper as follows. We first review related work, and then present the formal definition of our problem. After that, we give a detailed description of both ML-CBP and RIM. Finally, we evaluate ML-CBP and RIM in planning domains, and conclude the paper with future work.

## 2. Related work

Our work is related to case-based planning, model-lite planning, macro-operator based planning, and action model acquisition for planning. We will review these works in the following subsections.

### 2.1. Case-based planning

In case-based planning [8] a critical task is to efficiently identify problems that are most similar to the new problem to solve within a library of solved problems. Different case-based planners vary depending on how they adapt a solution to a new problem, whether or not using multiple existing plans for building a new solution, etc. [45]. As mentioned by Munoz-Avila and Cox [37], plan adaptation can be classified into two types, i.e., transformational and derivational analogy. In transformational analogy previous plans are reused in the new problem by making suitable changes [11,49]. Systems like CHEF [20] and PLEXUS [1] viewed the case library as an extensional representation of the domain knowledge. CHEF's use of case modification rules, for example, serves a similar purpose as our use of incomplete domain models. The post-CHEF case-based planning work largely focused on having access to a from-scratch planner operating on complete domain models (cf. [30,50]). The most recent of this line of work is OAKPlan [44], which we compare against. OAKPlan aims to retrieve planning cases from plan libraries which contain more than ten thousand cases, choose heuristically a suitable candidate and adapt it to provide a good quality solution plan given a planning problem (including complete action models) as input. A critical task in case-based planning is to efficiently identify, within a large library of already solved problems, those which are most similar to the new problem to solve. Vallati et al. [48] exploit new features that was first proposed by [15] to identify planning problems. The priar system [30] exploited three different types of validations, i.e., filtering condition, precondition, and phantom goal, as well as different reduction levels for the plan that represents a hierarchical decomposition of its structure, along with five different strategies for repairing validation failures. In contrast, the SPA system [21] uses the plan representation of causal links and order constraints. It performs the process of plan adaptation by a fairly simple extension of the process of plan generation. Plan generation in SPA is a special case of plan adaptation assuming there is no retrieved structure to exploit.

In derivational analogy, cases are derivational traces instead of plans as in transformation analogy, i.e., cases are sequences of computational steps a planner follows to generate plans, e.g., derSNLP [27], CAPlan/CbC [35], CLAM [33], Paris [4] and HICAP [36], etc. Since planners can replay derivational traces related to new problems, derivational analogy provides more flexibility, without requiring transformational operators. derSNLP and CAPlan/CbC build on a partial-order planner GPG [17], a planner using planning graphs. CLAM [33] constructs analogy-driven proof plans and uses derivational analogy to reformulate the source plans for case replay. CLAM exploits first-principles to achieve the unsolved goals. Paris [4] employs first-principles planning as well. The difference is that it stores and reuses abstract plans that use actions with aggregate granularity. HICAP is an interactive hierarchical CBP system [36], which extends retrieved cases via first-principle planning, using first principles to adapt the plan obtained during replay.

Our work however differs from previous work in two ways. First, unlike us allowing domain models being incomplete, previous work generally assumes access to a (more) complete domain model during its debugging stage. Second, previous work generally tries to adapt a specific case to the problem at hand, while our work expands a skeletal plan with relevant plan fragments mined from multiple library plans.

### 2.2. Model-lite planning

Planning with incomplete information dates back to the beginning of 90s [41,14], where incomplete information about planning states was first introduced. To complement the incomplete information of states, planning with observations were introduced, resulting in planning with conditional effects [43,40]. Recently planning with incomplete information has also

extended to multi-agent environments [47]. Despite the success of previous systems, they all assume the domain model or action description is complete.

Planning with risks is similar to planning with incomplete information [7], where the action descriptions or domain models, instead of states, are incomplete. The action incompleteness could be modeled as state incompleteness and plans might be conformant. Conformant planning techniques are, however, not readily applicable because plans with risks cannot guarantee goal satisfaction, rather, only partial satisfaction. Instead of compiling incomplete action descriptions to incomplete states, symbolic constraint propagation and model counting based approaches have been proposed for counting the number of incomplete domain model interpretations under which a plan is consistent [34].

The recent focus on planning with incomplete domain models originated with the work on "model-lite planning" [29]. Approaches for model-lite planning must either consider auxiliary knowledge sources or depend on long-term learning. While our work views the case-library as the auxiliary knowledge source, work by Nguyen et al. [39] and Weber et. al. [10] assume that domain writers are able to provide annotations about missing preconditions and effects. It would be interesting to see if these techniques can be combined with ours. One interesting question, for example, is whether the case library can be compiled over time into such possible precondition/effect annotations.

### 2.3. Macro-operator based planning

There have been many planning approaches that learn and use macro-operators in planning. Fikes et al. [16] proposed to build generalized plans and use them as macro actions and monitor plan execution. Korf [31] proposed to learn efficient strategies for solving difficult problems by searching macro-operators. Iba [26] developed a system, called MACLEARN, to learn new macro operators, which can be defined based on other macro operators, to improve the efficiency of problem solving. Botea et al. [9] presented and compared two automated methods that learn relevant information from previous experience in a domain and use it to solve new problem instances, by lifting partial-order macros from plans based on an analysis of causal links between successive actions. Newton et al. [38] proposed an offline method that learns macros genetically from plans for arbitrarily chosen planners and domains without focusing on exploiting particular planner or domain properties. He et al. [22] presented a POMDP algorithm for planning under uncertainty with macro-actions (PUMA) that automatically constructed and evaluated open-loop macro-actions within forward-search planning.

All these efforts however assume that the learner has access to a complete domain model, and are motivated mainly by the desire to reduce the time taken for planning. In contrast, `RIM` culls and uses macro-operators to increase the accuracy of the incomplete model.

### 2.4. Action model learning

A third strand of research that is also related to our work is that of action model learning. Work such as [52,60,59,56,55,54,51,13,18] focuses on learning action models directly from observed (or pre-specified) plan cases. The connection between this strand of work and our work can be seen in terms of the familiar up-front vs. demand-driven knowledge transfer: the learning methods attempt to condense the case library directly into STRIPS models before using it in planning, while we transfer knowledge from cases on a per-problem basis. In contrast, work such as [2], as well as much of the reinforcement learning work [46] focuses on learning models from trial-and-error execution.[1] This too can be complementary to our work in that execution failures can be viewed as opportunities to augment the case-library (cf. [28]).

We focus on learning in the presence of an existing incomplete model. Starting from STRIPS [16], macro-operator learning has been a staple in automated planning (cf. [31,26,12,9,38]). All these efforts however assume that the learner has access to a complete domain model, and are motivated mainly by the desire to reduce the time taken for planning. In contrast, we cull and use macro-operators to increase the accuracy of the incomplete model, which takes a global view of first refining the incomplete model using the learned models of primitive and macro actions.

## 3. Problem definition

A STRIPS domain model 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 $\mathcal{R}_{\mathcal{O}}$ as the set of propositions instantiated from $\mathcal{R}$ with respect to a set of typed objects $\mathcal{O}$. Given $\mathcal{M}$ and $\mathcal{O}$, we define a planning problem as $\mathcal{P} = \langle \mathcal{O}, s_0, g \rangle$, where $s_0 \subseteq \mathcal{R}_{\mathcal{O}}$ is an initial state, $g \subseteq \mathcal{R}_{\mathcal{O}}$ 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 achieve goal $g$ starting from $s_0$.

We define a STRIPS domain model to be complete with respect to a set of predefined solutions of planning problems (i.e. cases), if it can be used (by a planner) to generate that set of predefined solutions to the planning problems. An action model $\langle a, \text{PRE}(a), \text{ADD}(a), \text{DEL}(a) \rangle$ is considered *incomplete* if in $\text{PRE}(a)$, $\text{ADD}(a)$, or $\text{DEL}(a)$ there are predicates missing

[1] This latter has to in general be limited to ergodic domains.

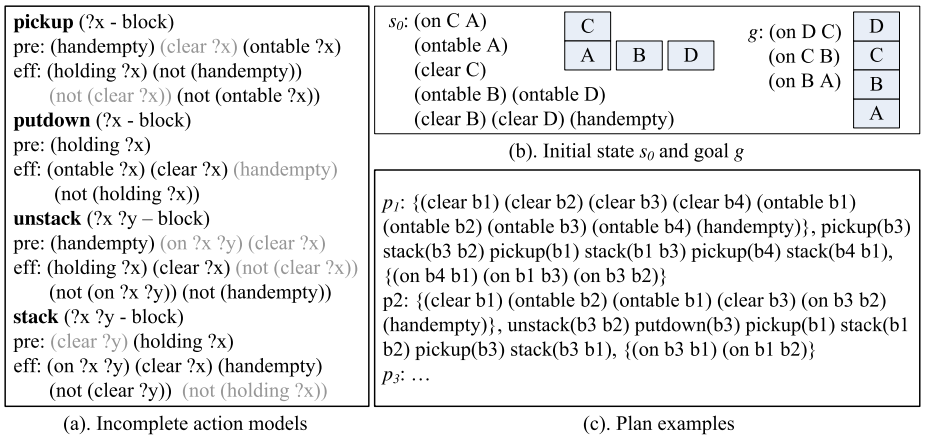


**Fig. 3.** An input example of `ML-CBP` for domain *blocks*.

which lead to some of the set of predefined planning solutions to fail. We denote $\widetilde{\mathcal{A}}$ as a set of incomplete action models, and thus $\widetilde{\mathcal{M}} = \langle \mathcal{R}, \widetilde{\mathcal{A}} \rangle$ the corresponding incomplete STRIPS domain. Although action models in $\widetilde{\mathcal{A}}$ might have incomplete preconditions and effects, we assume that the number of action models is identical between $\widetilde{\mathcal{A}}$ and $\mathcal{A}$, i.e., $|\widetilde{\mathcal{A}}| = |\mathcal{A}|$, and that preconditions and effects specified in $\widetilde{\mathcal{A}}$ are correct. We are now ready to formally state the problem we address:
**Given:** $\langle \widetilde{\mathcal{P}}, \widetilde{\mathcal{M}}, \mathcal{C} \rangle$, where $\widetilde{\mathcal{P}} = \langle \mathcal{O}, s_0, g \rangle$ is a planning problem, $\widetilde{\mathcal{M}}$ is an incomplete domain available to the planner, and $\mathcal{C}$ is a set of successful solution plans (or plan cases) that are correct with respect to the complete model $\mathcal{M}$. Specifically, each plan case provides a plan $p_i$ for problem $\mathcal{P}_i = \langle \mathcal{O}^i, s_0^i, g^i \rangle$ that is correct with respect to $\mathcal{M}$. A plan case $c_i$ is composed of a triple $c_i = \langle s_0^i, p_i, g^i \rangle$.
**Objective:** Find a solution to $\widetilde{\mathcal{P}}$ that is correct w.r.t. $\mathcal{M}$.

We note that $\mathcal{M}$ is not given directly but only known indirectly (and partially) in terms of the cases that are successful in it. We note that since the incomplete model $\widetilde{\mathcal{M}}$ is not necessary an abstraction of the complete domain, a plan correct w.r.t. $\mathcal{M}$ might not even be correct with respect to $\widetilde{\mathcal{M}}$ (in other words, the upward refinement property [3] does not hold with model incompleteness).

An example input of our planning problem in *blocks*[2] domain is shown in Fig. 3. It is composed of three parts: (a) incomplete action models, (b) the problem including the initial state $s_0$ and goals $g$, and (c) a set of plan cases. In Fig. 3(a), the gray parts indicate the missing predicates. In Fig. 3(c), $p_1$ and $p_2$ are two plan cases with the initial states and goals in brackets. One solution to the example problem is "`unstack(C A) putdown(C) pickup(B) stack(B A) pickup(C) stack(C B) pickup(D) stack(D C)`".

## 4. The `ML-CBP` algorithm

**Algorithm 1.** The `ML-CBP` algorithm.

**Input:** $\widetilde{\mathcal{P}}$, $\widetilde{\mathcal{M}}$, and a set of plan cases $\mathcal{C}$.
**Output:** the plan $p^{sol}$ for solving the problem.
1: generate a set of causal pairs $\mathcal{L}$ with $\widetilde{\mathcal{P}}$ and $\widetilde{\mathcal{M}}$;
2: build a set of plan fragments $\varphi$:
$\varphi = build\_fragments(\widetilde{\mathcal{P}}, \mathcal{C})$;
3: mine a set of frequent plan fragments $\mathcal{F}$:
$\mathcal{F} = freq\_mining(\varphi)$;
4: $p^{sol} = concat\_frag(\mathcal{L}, \mathcal{F})$;
5: **return** $p^{sol}$;

An overview of our `ML-CBP` algorithm can be found in Algorithm 1. We first use $\widetilde{\mathcal{P}}$ and $\widetilde{\mathcal{M}}$ to generate a skeletal plan represented by a set of causal pairs. After that, we build a set of plan fragments based on plan cases and causal pairs, and then mine a set of frequent plan fragments with a specific threshold. These frequent fragments will be integrated together to form the final solution $p^{sol}$ based on causal pairs. Next, we describe each step in detail.

### 4.1. Generate causal pairs

Given the initial state $s_0$ and goal $g$, we generate a set of causal pairs $\mathcal{L}$. A causal pair is an action pair $\langle a_i, a_j \rangle$ such that $a_i$ provides one or more conditions for $a_j$. The procedure to generate $\mathcal{L}$ is shown in Algorithm 2. Note that, in step 3 of

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

**Algorithm 2.** Generate causal pairs.

**input:** $\widetilde{\mathcal{P}} : \langle \mathcal{O}, s_0, g \rangle$, and $\widetilde{\mathcal{M}}$.
**output:** a set of causal pairs $\mathcal{L}$.

```
1: L = ∅;
2: for each proposition f ∈ g do
3:     generate a plan with M̃, denoted by a set of causal pairs L′, to transit s_0 to f;
4:     L = L ∪ L′;
5: end for
6: return L;
```

Algorithm 2, $\mathcal{L}'$ is an empty set if $f$ cannot be achieved. In other words, skeletal plans may not provide any guidance for some top level goals. Actions in causal pairs $\mathcal{L}$ is viewed as a set of *landmarks* for helping construct the final solution, as will be seen in the coming sections. In this step we do not consider the interaction among goals, since there is no available information that we can exploit to capture the interaction of goals based on the input plan cases or incomplete action models. We thus at first build a "skeletal" plan based on single goals, ignoring the interaction among goals, and then fix or revise the skeletal plan using frequent plan fragments.

**Example 1.** As an example, causal pairs generated for the planning problem given in Fig. 3 is {⟨ `pickup(B), stack(B A)` ⟩, ⟨`unstack(C A), stack(C B)`⟩, ⟨ `pickup(D), stack(D C)`⟩}.

### 4.2. Creating plan fragments

In the procedure *build_fragments* of Algorithm 1, we would like to build a set of plan fragments $\varphi$ by building mappings between "objects" involved in $s_0$ and $g$ of $\widetilde{\mathcal{P}}$ and those in $s_0^i$ and $g^i$ of plan case $p_i \in \mathcal{C}$. In other words, a mapping, denoted by $m$, is composed of a set of pairs $\{\langle o', o \rangle\}$, where $o$ and $o'$ are objects in $\widetilde{\mathcal{P}}$ and $p_i$ respectively. We can apply mapping $m$ to a plan case $p_i$, whose result is denoted by $p_i|_m$, such that $s_0^i|_m$ and $s_0$ share common propositions, likewise for $g^i$ and $g$. We measure a mapping $m$ by the number of propositions shared by initial states $s_0^i|_m$ and $s_0$, and goals $g^i$ and $g$, assuming that all propositions are "equally" important in describing states. We denote the number of shared propositions by $\theta(p_i, m)$, i.e.,

$$\theta(p_i, m) = |(s_0^i|_m) \cap s_0| + |(g^i|_m) \cap g|.$$

**Example 2.** In Fig. 3, a possible mapping $m$ between $\langle s_0, g \rangle$ and $\langle s_0^1, g^1 \rangle$ of $p_1$ is $\{\langle b4, D\rangle, \langle b1, C\rangle, \langle b3, B\rangle, \langle b2, A\rangle\}$. The result of applying $m$ to $s_0^1$ is $s_0^1|_m = \{$`(clear C)(clear A)(clear B)(clear D)(ontable C)(ontable A)(ontable B)(ontable D)(handempty)`$\}$. $g^1|_m$ can be computed similarly, and thus $\theta(p_1, m) = |(s_0^1|_m) \cap s_0| + |(g^1|_m) \cap g| = 10$.

Given that there might be different mappings between $\widetilde{\mathcal{P}}$ and $p_i$, we seek for the one maximally mapping $p_i$ to $\widetilde{\mathcal{P}}$, defined as $m^* = \arg\max_m \theta(p_i, m)$. The more common propositions $\widetilde{\mathcal{P}}$ and $p_i$ share, the more "similar" they are. Note that mappings between objects of the same types are subject to the constraint that they should have the set of "features" in the domain, defined by unary predicates of the corresponding types. For instance, "b3" can be mapped to "B" in our running example since both of them are the two blocks having the same features "on table" and "clear" in the two problems. In practice, we find that this requirement significantly reduces the amount of mappings that need to be considered, actually allowing us to find $m^*$ in a reasonable running time. It is true that the problem of building mappings is NP-hard. To handle this problem we have to leverage constraints to prune the explosion space and/or exploit approximate algorithms, such as greedy methods, to estimate the maximal mapping. In the experiment, we find that the above-mentioned constraints can indeed reduce a great deal of the mapping space, and thus do not consider exploiting approximate approaches in the procedure of *build_fragments*. We believe it is straightforward to utilize approximate approaches to further reducing the space when the problem becomes huge.

We apply the mapping $m^*$ to $p_i$ to get a new plan $p_i|_{m^*}$. We then scan the new plan to extract subsequences of actions such that all objects in each subsequence appear in the given problem $\widetilde{\mathcal{P}}$. We call these subsequences *plan fragments*. We repeat the process for plan cases $\mathcal{C}$ to obtain the set $\varphi$ of plan fragments.

**Example 3.** In Example 2, we find that for $p_1$, $m^* = \{\langle b4, D\rangle, \langle b1, C\rangle, \langle b3, B\rangle, \langle b2, A\rangle\}$. Thus, $p_1|_{m^*}$ is "`pickup(B) stack(B A) pickup(C) stack(C B) pickup(D) stack(D C)`". For $p_2$ in Fig. 3, $m^*$ is $\{\langle b3, C\rangle, \langle b1, B\rangle, \langle b2, A\rangle\}$. Thus, $p_2|_{m^*}$ is "`unstack(C A) putdown(C) pickup(B) stack(B A) pickup(C) stack(C B)`". Both of the two are plan fragments.

### 4.3. Mining frequent plan fragments

In step 3 of Algorithm 1, we aim at building a set of frequent plan fragments $\mathcal{F}$ using the procedure *freq_mining*. Those are plan fragments occurring multiple times in different plan cases, increasing our confidence on reusing them in solving the new problem. We thus borrow the notion of frequent patterns defined in [53,42] for extracting $\mathcal{F}$.

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 $s = \langle s_1, s_2, \ldots, s_m \rangle$. 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 $s_b = \langle b_1, b_2, \ldots, b_m \rangle$, denoted by $s_a \sqsubseteq s_b$, if there exist integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \ldots, a_n \subseteq b_{i_n}$. 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, the frequent sequential pattern mining problem is to find the complete set of sequential patterns whose support is larger than the threshold.

We view each action of plan fragments as an itemset, and a plan fragment as a sequence, which suggests plan fragments can be viewed as a sequence database. Note that in our case an itemset has only one element, and the indices of those in the subsequence are continuous. We fix a threshold $\delta$ and use the SPADE algorithm [53] to mine a set of frequent patterns. There are many frequent patterns which are subsequences of other frequent patterns. We eliminate these "subsequences" and keep the "maximal" patterns, i.e., those with the longest length, as the final set of frequent plan fragments $\mathcal{F}$.

**Example 4.** In Example 3, if we set $\delta$ to be 2 and 1, the results are shown below (frequent plan fragments are partitioned by commas):

**Plan Fragments:**
#1."`pickup(B) stack(B A) pickup(C) stack(C B) pickup(D) stack(D C)`"
#2."`unstack(C A) putdown(C) pickup(B) stack(B A) pickup(C) stack(C B)`"
**Frequent Plan Fragments $\mathcal{F}$ ($\delta = 2$):**
{`pickup(B) stack(B A) pickup(C) stack(C B)`}
**Frequent Plan Fragments $\mathcal{F}$ ($\delta = 1$):**
{`pickup(B) stack(B A) pickup(C) stack(C B) pickup(D) stack(D C), unstack(C A) putdown(C) pickup(B) stack(B A) pickup(C) stack(C B)`}

Note that the following frequent patterns are eliminated when $\delta = 2$ (likewise when $\delta = 1$): {`pickup(B), stack(B A), pickup(C), stack(C B), pickup(B) stack(B A), stack(B A) pickup(C), pickup(C) stack(C B), pickup(B) stack(B A) pickup(C), stack(B A) pickup(C) stack(C B)`}.

### 4.4. Generating final solution

In step 4 of Algorithm 1, we generate the final solution using the sets of causal pairs and frequent plan fragments generated in the previous steps. We address the procedure *concat_frag* by Algorithm 3. In Algorithm 3, we scan each causal pair in $\mathcal{L}$ and each frequent plan fragment in $\mathcal{F}$. If a plan fragment contains an action (or both actions) of a causal pair, we append the plan fragment to the final solution $p^{sol}$ and remove all the causal pairs that are satisfied by the new $p^{sol}$. We repeat the procedure until the solution is found, i.e., $\mathcal{L} = \emptyset$, or no solution is found, i.e., the procedure returns *false*.

**Algorithm 3.** $p^{sol} = concat\_frag(\mathcal{L}, \mathcal{F})$.

**input:** a set of causal pairs $\mathcal{L}$, and a set of frequent fragments $\mathcal{F}$;
**output:** The solution plan $p^{sol}$.

```
1: while L ≠ ∅ do
2:     randomly select a pair ⟨a_i, a_j⟩ ∈ L;
3:     randomly select some f ∈ F, such that:
       (a_i ∈ f ∨ a_j ∈ f) and share(p^sol, f) = true;
4:     if there is no such f, return ⟨⟩;
5:     p^sol = append(p^sol, f);
6:     L = removelinks(p^sol, L);
7: end while
8: return p^sol;
```

In step 3 of Algorithm 3, we randomly select a plan fragment $f$ such that $f$ contains actions $a_i$ and $a_j$ and shares a common action subsequence with $p^{sol}$. Note that the procedure *share* returns *true* if $p^{sol}$ is empty or $p^{sol}$ and $f$ share a common action subsequence. That is to say, two plan fragments are concatenated only if they have some sort of *connection*,

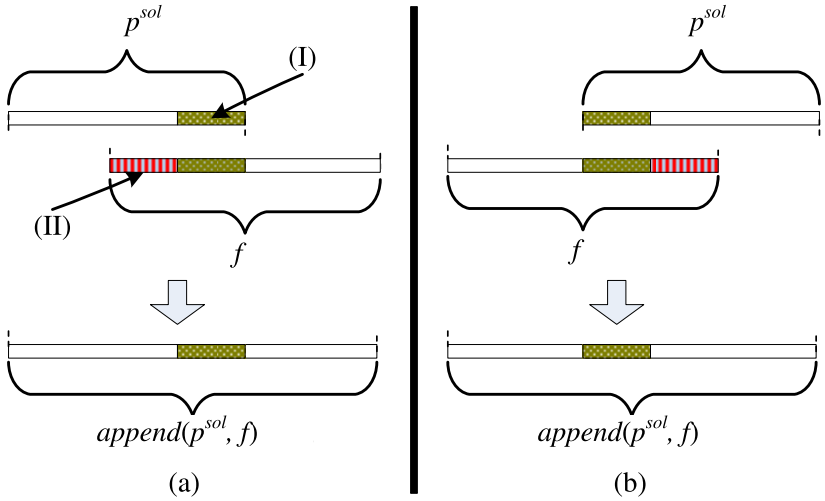


**Fig. 4.** (a) $f$ is concatenated at the end of $p^{sol}$; (b) $f$ is concatenated at the beginning of $p^{sol}$; Part (I) is the maximal action subsequence; Part (II) is the action subsequence that is different from $p^{sol}$.

which is indicated by common action subsequence. In step 4, we return an empty plan if there is no plan fragment satisfying a single causal pair $f$. This is because we require that the output solution should satisfy all the causal pairs $\mathcal{L}$ generated based on a single proposition in the goal (a set of propositions). This is reasonable since the causal pairs $\mathcal{L}$ is a subset of causal pairs generated based on the final correct solution (corresponding to the whole goal). If there is no plan fragment in the set of frequent fragments $\mathcal{F}$ satisfying one of the causal pair in $\mathcal{L}$, we can conjecture that no desired solution will be found based on fragments $\mathcal{F}$. That is why we output an empty solution when there is no plan fragment satisfying a single causal pair, without considering the remaining pairs in $\mathcal{L}$. In step 5 of Algorithm 3, we concatenate $p^{sol}$ and $f$ based on their maximal common action subsequence, which is viewed as the strongest connection between them. Note that the common action subsequence should start from the beginning of $p^{sol}$ or end at the end of $p^{sol}$. In other words, $f$ can be concatenated at the end of $p^{sol}$ or at the beginning, as is shown in Fig. 4. In step 6 of Algorithm 3, the procedure *removelinks* removes all causal pairs in $\mathcal{L}$ that are "satisfied" by $p^{sol}$. Example 5 demonstrates how this procedure works.

**Example 5.** In Example 4, we have two frequent plan fragments by setting $\delta = 1$. We concatenate them with causal pairs in Example 1. The result is shown as follows.

| |
|---|
| fragment 1: **pickup(B) stack(B A) pickup(C) stack(C B)** pickup(D) stack(D C) |
| fragment 2: unstack(C A) putdown(C) **pickup(B) stack(B A) pickup(C) stack(C B)** |
| result: unstack(C A) putdown(C) **pickup(B) stack(B A) pickup(C) stack(C B)** pickup(D) stack(D C) |
| solution: unstack(C A) putdown(C) **pickup(B) stack(B A) pickup(C) stack(C B)** pickup(D) stack(D C) |

The boldfaced part is the actions shared by fragments 1 and 2. The concatenating result is shown in the third row. After concatenating, we can see that all the causal pairs in $\mathcal{L}$ are satisfied and will be removed according to step 6 of Algorithm 3. The result is shown in the fourth row.

### 4.5. Discussion

The ML-CBP algorithm mainly functions by two steps: generating a skeletal plan (Algorithm 2) and refining the skeletal plan with frequent plan fragments (Algorithm 3) to produce the final solution. The first step aims to guide a plan to reach the goal by a skeletal plan (or a set of causal pairs), while the second step aims to fill "details" in the skeletal plan. We are aware that there might be "negative interactions" plan fragments that delete goals that have been reached by previous plan fragments and lead to failure solutions as a result. However, in step 3 of Algorithm 1, we filter these negative plan fragments by setting a frequency threshold, assuming negative plan fragments have low frequencies. Furthermore, when concatenating selected frequent fragments, we consider the maximally shared common actions between fragments, which indicates strongest connections of fragments, to further reduce the impact of negative fragments.

We note that ML-CBP does not check whether the solution is correct with respect to the incomplete model. As mentioned in the problem definition section, the rationale for this is that correctness with respect to the incomplete domain model is *neither a necessary nor a sufficient condition* for the correctness with respect to the (unknown) complete model. Indeed, enforcing the satisfaction with respect to incomplete domains can even prune correct solutions in some scenarios. For example, a precondition $p$ of action $a_i$ in a correct solution may not be added by any previous action $a_j$ (assuming $p$ is not in the initial state) since $p$ is missing in the add lists of all $a_j$. In the other hand, directly taking a similar solution from the plan library as the solution to the target problem, as done by OAKPlan without the repair phase, is improper in the sense that the target problem is not exactly the same as any of the plan library.

## 5. 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 cases $\mathcal{C}$ 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 4. It begins with the step of collecting sets of predicates $P$, action schemas $A$ and macro-operator schemas $O$ from incomplete action models $\tilde{\mathcal{A}}$ and plan cases $\mathcal{C}$. 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 cases 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 4.** Phase I: refining domain models.

**Input:** incomplete action models $\tilde{\mathcal{A}}$, and plan cases $\mathcal{C}$.
**Output:** macro-operators $\mathcal{O}$ and action models $\mathcal{A}$.
1: build sets of predicates, action schemas and macro-operator schemas: $(P, A, \mathcal{O}) = build\_schemas(\tilde{\mathcal{A}}, \mathcal{C})$;
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 $\mathcal{A}$;

### 5.1. 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 $\tilde{\mathcal{A}}$, and view them as the initial set of predicates $P$. We then scan each proposition in plan cases 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 cases.

There are no easy ways to construct macro-operator schemas, since they neither exist in the plan cases 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 cases, 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 $\delta$ for frequency and a threshold $\theta$ for strength. We borrow the notion of frequent patterns defined in [53,42] to mine the frequent action subsequences, as done by Section 4.3. We convert the set of plan cases 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 cases as an itemset, which has only one element, and a plan case as a sequence. The set of plan cases can now be

**Table 1**
The example macro-operator schemas.

```
(:macro macro1
(:parameters ?x - block ?y - block ?z - block)
(:actions (putdown ?x) (unstack ?y ?z) (stack ?y ?x)))

(:macro macro2
(:parameters ?x - block ?y - block ?z - block ?w - block)
(:actions (putdown ?x) (unstack ?y ?z) (stack ?y ?w)))
```

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 [53], to mine a set of frequent action 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 cases, 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.

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 $\mathcal{C}$, whose frequencies are larger than the preset support constant $\delta$, neglecting the parameters of actions in $\mathcal{C}$.
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 $\delta$ and whose strengths are larger than the preset constant $\theta$, 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)`".

### 5.2. 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.

#### 5.2.1. State constraints

We first build soft constraints encoding possible preconditions and effects of actions and macro-operators implied by state transitions in plan cases. We preprocess plan cases using incomplete action models to obtain more state information for building state constraints. To do this, we simply "execute" each plan case starting from its initial state, and calculate (incomplete) states between actions using incomplete action models. In particular, given a plan case $t = \langle s_0, a_1, \ldots, s_{n-1}, a_n, g\rangle$, we execute $t$ from $s_0$ using the incomplete action models $\tilde{\mathcal{A}}$, 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).[3] 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 cases by $\mathcal{C}'$.

By observation, in $\mathcal{C}'$ 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[4]:

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] 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.

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

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 cases in $\mathcal{C}'$ and count the occurrences of each constraint in $\mathcal{C}'$. We assign the number of occurrences as the weight of the corresponding constraint.

#### 5.2.2. 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 4, $a_i$ 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.

### 5.3. Building hard constraints

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

#### 5.3.1. 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.

#### 5.3.2. 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 [52] 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:

$$p \in \text{DEL}(a) \Rightarrow p \in \text{PRE}(a).$$

#### 5.3.3. Plan constraints

We require that the action models learned do not violate the correctness of plan cases. This requirement is imposed on the relationship between ordered actions in plan cases, ensuring that the causal links in the plan cases are not broken. That is, for each precondition $p$ of an action $a_j$ in a plan case, 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:

$$\underset{p \in \text{PRE}(a_j)}{\forall} \Big\{ p \in s_0 \wedge \underset{0<k<j}{\forall} p \notin \text{DEL}(a_k) \Big\},$$

or

$$\underset{p \in \text{PRE}(a_j)}{\forall} \Big\{ \underset{0<i<j}{\exists} [p \in \text{ADD}(a_i) \wedge \underset{i<k<j}{\forall} p \notin \text{DEL}(a_k)] \Big\}.$$

*5.3.4. 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

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

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 $w_{max}$.

### 5.4. Solving constraints

We put all constraints together and solve them with a weighted MAX-SAT solver [32]. We exploit MaxSatz [32] 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 $a$.

### 5.5. 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[5] [24,25]. 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 $\tilde{\mathcal{P}}$.

## 6. Experiments

In this section we present an extensive empirical evaluation. In the following, we describe the experimental setup, and then discuss the results of our experiments.

### 6.1. Dataset and criterion

We evaluate `ML-CBP` and `RIM` in five planning domains: *Blocks*,[2] *Driverlog*,[6] *Depots*,[6] Child-Snack[7] and Barman,[7] which are briefly introduced as follows.

- Blocks: The objects in the domain include a finite number of cubical blocks, and a table large enough to hold all of them. Each block is on a single other object (either another block or the table). For each block, either it is clear or else there is a unique block a sitting on it. There is one kind of action: move a single clear block, either from another block onto the table, or from an object onto another clear block. As a result of moving block b from c onto d, b is sitting on d instead of c, c is clear (unless it is the table), and d is not clear (unless it is the table) [19].
- Driverlog: This domain has drivers that can walk between locations and trucks that can drive between locations. Walking requires traversal of different paths from those used for driving, and there is always one intermediate location on a footpath between two road junctions. The trucks can be loaded or unloaded with packages (with or without a driver present) and the objective is to transport packages between locations, ending up with a subset of the packages, the trucks and the drivers at specified destinations.
- Depots: The domain consists of actions to load and unload trucks, using hoists that are available at fixed locations. The loads are all crates that can be stacked and unstacked onto a fixed set of pallets at the locations. The trucks do not hold crates in a particular order, so they can act like a table in the Blocks domain, allowing crates to be reordered.
- Child-Snack: This domain is to plan how to make and serve sandwiches for a group of children in which some are allergic to gluten. There are two actions for making sandwiches from their ingredients. The first one makes a sandwich and the second one makes a sandwich taking into account that all ingredients are gluten-free. There are also actions to put a sandwich on a tray and to serve sandwiches. Problems in this domain define the ingredients to make sandwiches at the initial state. Goals consist of having all kids served with a sandwich to which they are not allergic.

[5] http://fai.cs.uni-saarland.de/hoffmann/ff.html.
[6] http://ipc02.icaps-conference.org.
[7] https://helios.hud.ac.uk/scommv/IPC-14/domains_sequential.html.

- Barman: In this domain there is a robot barman that manipulates drink dispensers, glasses and a shaker. The goal is to find a plan of the robot's actions that serves a desired set of drinks. In this domain deletes of actions encode relevant knowledge given that robot hands can only grasp one object at a time and given that glasses need to be empty and clean to be filled.

In each domain, we generate 400 plan cases using a classical planner such as FF and solve 100 new planning problems based on different percentages of completeness of domain models. For example, we use $\frac{4}{5}$ to indicate one predicate is missing among five predicates of the domain. We generate new problems and plan cases using problem generators with a random number of objects, respectively. The random number was set from 10 to 50. Note that objects (or object symbols) used in plan cases are completely different from those used in testing problems; thus plan cases and new problems do not share common propositions.

We measure the accuracy of ML-CBP and RIM as the percentage of correctly solved planning problems. Specifically, we exploit ML-CBP (or RIM) to generate a solution to a planning problem, and execute the solution from the initial state to the goal. If the solution can be successfully executed starting from the initial state, and the goal is achieved, then the number of correctly solved problems is increased by one. The accuracy, denoted by $\lambda$, can be computed by $\lambda = \frac{N_c}{N_t}$, where $N_c$ is the number of correctly solved problems, and $N_t$ is the number of total testing problems. Note that when testing the accuracy of ML-CBP and RIM, we assume that we have *complete* domain models available for executing generated solutions.

In the following subsections, we evaluate ML-CBP and RIM in the following aspects:

1. We compare ML-CBP and RIM to state-of-the-art case based planning algorithm OAKPlan to see the advantages/disadvantages of ML-CBP, RIM and OAKPlan, with respect to different number of plan cases.
2. We also compare ML-CBP, RIM and OAKPlan to see the impact of completeness of action models.
3. To see the impact of threshold in ML-CBP and RIM, we compare ML-CBP and RIM by varying the value of threshold from 5 to 25.
4. To see the advantage of learnt macro-operators in RIM, we compare RIM to ARMS by varying the number of plan cases.
5. To see the efficiency of ML-CBP and RIM, we show the running time of both ML-CBP and RIM with respect to different number of plan cases.

### 6.2. *Performance of* ML-CBP, RIM *and OAKPlan w.r.t. plan cases*

We compared ML-CBP and RIM to the state-of-the-art case-based planning system OAKPlan [44]. All the three algorithms are given the same (incomplete) model as their input. We note that OAKPlan, like most recent case-based planners, assumes a complete model (and is thus more of a plan reuse system rather than a true case-based planning system). Nevertheless, given that it is currently considered the state-of-the-art case-based planner, we believe that comparing its performance with ML-CBP and RIM in the context of incomplete models is instructive as it allows us to judge them with respect to the original motivation for case-based planning–which is to make up for the incompleteness of the model with the help of cases. We set the percentage of completeness to be a modest value, i.e., 60%, and the threshold $\delta$ to be 15 which was verified to be the value with the best performance (as shown in Table 2) for both ML-CBP and RIM, and the threshold for strength of all macro-operators $\theta$ to be a modest value, i.e., 0.25, for RIM. Note that $\delta$ is a threshold of frequency of action subsequences to filter action subsequences with low frequency, as defined in Section 4.3, and $\theta$ is a threshold of strength of macro-operators to filter macro-operators, as defined in Section 5.1. We varied the number of plan cases from 40 to 400 and run ML-CBP and RIM to solve 100 planning problems. Fig. 5 shows the accuracy $\lambda$ with respect to the number of plan cases used.

From Fig. 5, we found that accuracies of ML-CBP, RIM and OAKPlan generally became larger when the number of plan cases increased. This is consistent with our intuition, since there is more knowledge to be used when plan cases become larger. We also found that both ML-CBP and RIM generally had higher accuracy than OAKPlan in all the five domains. This is because ML-CBP (or RIM) exploits the information of incomplete models to mine *multiple* high quality plan fragments, i.e., ML-CBP (RIM) integrates the knowledge from both incomplete models and plan cases, which may help each other, to attain the final solution. In contrast, OAKPlan first retrieves a case, and then adapts the case using the inputted incomplete model, which may fail to make use of valuable information from other cases (or plan fragments) when adapting the case.

Comparing the curves of ML-CBP and RIM, we can see that the accuracy of ML-CBP is generally smaller than RIM at the beginning, and becomes larger than RIM as the number of cases increases (e.g., larger than 140). The rationale behind this result is that ML-CBP solves model-lite planning problems based on aligning plan fragments (rather than action models, as done by RIM). The higher quality the plan fragments are of, the larger the accuracy of ML-CBP is. Since the quality of plan fragments is dependent on the number of plan cases, it sharply increases when the number of plan cases increases. In contrast, RIM is less dependent on the number of plan cases (more reliant on action models) compared to ML-CBP. The curve of ML-CBP thus rises more sharply than RIM when the number of plan cases increases. From Fig. 5 we observe that the accuracy of ML-CBP is generally larger than RIM after 140 in the first three domains (after 240 in the last two domains, i.e., *child-snack* and *barman*). From the trends of the curves we can see that the accuracy of RIM will not be able to catch up ML-CBP in all five domains. By observation, we also find that the accuracy of ML-CBP (or RIM) is no less than 0.8 when

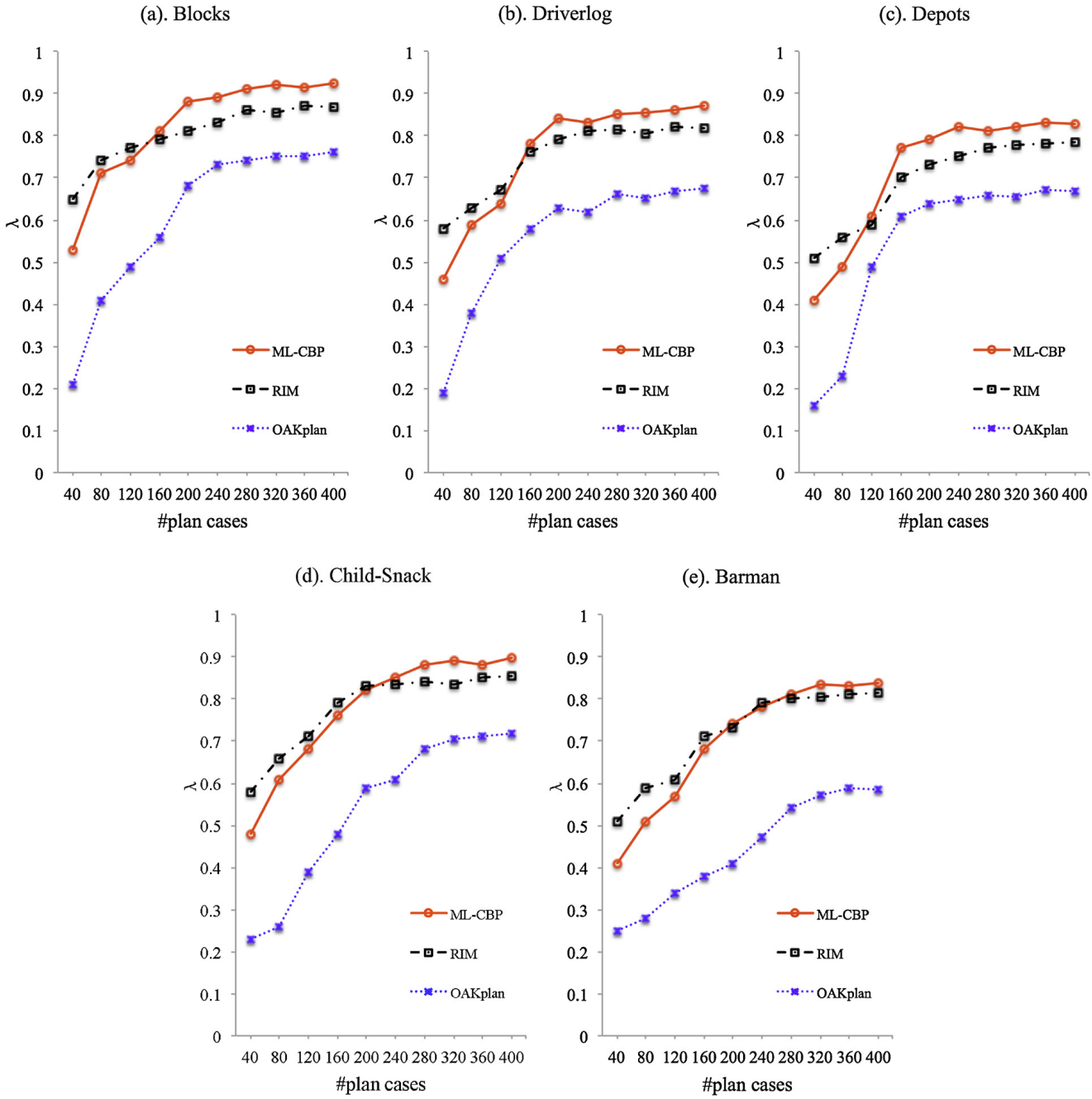


**Fig. 5.** Accuracy w.r.t. number of plan cases.

the number of plan cases is more than 160 in domains *blocks*, *driverlog*, *depots*, and *child-snack* (it is no less than 0.7 when the number of plan cases is more than 240 in domain *barman*).

### 6.3. *Performance of* ML-CBP, RIM *and OAKPlan w.r.t. percentage of completeness*

To test the change of accuracies with respect to different degrees of completeness, we varied the percentage of completeness from 20% to 100%, and ran ML-CBP with 200 plan cases by setting $\delta = 15$. We also compared the accuracy with OAKPlan. The result is shown in Fig. 6.

We found that all accuracies of ML-CBP, RIM, and OAKPlan increased when the percentage of completeness increased, due to more information provided when the percentage increasing. This is consistent with our intuition, since the more the information is provided, the more the knowledge is available for helping improve the accuracies. When the percentage is 100%, ML-CBP, RIM and OAKPlan can solve all the solvable planning problems successfully, since all planning problems can be solved similar to classical planning by all three systems given the action models are fully specified. Similar to Fig. 5,

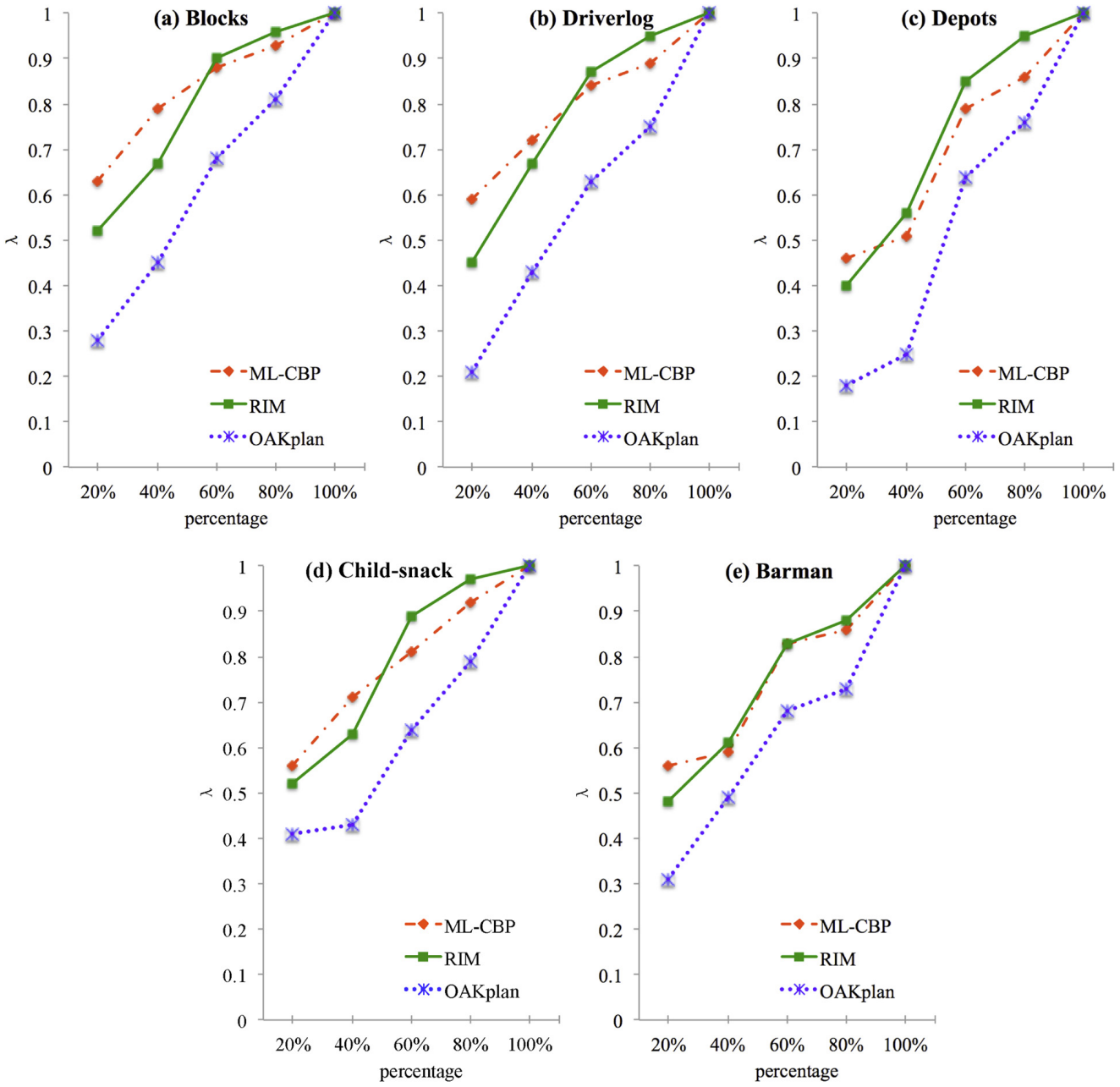


**Fig. 6.** Accuracy w.r.t. percentage of completeness.

both ML-CBP and RIM perform better than OAKPlan. The reason is similar to Fig. 5, i.e., simultaneously exploiting both knowledge from incomplete domain models and plan cases could be helpful.

Looking at the curves of ML-CBP and RIM, we can see that ML-CBP performs better than RIM at the beginning (e.g., percentage smaller than 60% in domains *blocks* and *driverlog*), but becomes worse than RIM when the percentage of completeness increases. This fact reveals that the case-based approach, i.e., ML-CBP, performs much better than the model-based approach, i.e., RIM, when the percentage of completeness of action models is small. The rationale behind this fact is that, since the model-based approach (i.e., RIM) solves the model-lite planning problems directly based on learnt action models (and macro-operators as well), the higher the quality of the input action models are (which largely influences the quality of the learnt action models), the better the accuracy of the model-based approach is. On the other hand, the case-based approach (i.e., ML-CBP) solves the model-lite planning problems based on aligning plan fragments instead of the learnt action models, which in some sense indicates the case-based approach is less dependant on the completeness of input action models compared to the model-based approach. This is why the curve of the case-based approach ML-CBP less sharply increases than the model-based approach RIM when the completeness of action models increases. From Fig. 6 we can see that the accuracy of RIM generally becomes better than ML-CBP after the completeness of action models exceeds 60%.

**Table 2**
Accuracy with respect to different thresholds.

| | Threshold | Blocks | Driverlog | Depots | Child-snack | Barman |
|---|---|---|---|---|---|---|
| ML-CBP | $\delta = 5$ | 0.80 | 0.78 | 0.73 | 0.82 | 0.69 |
| | $\delta = 15$ | **0.88** | **0.84** | 0.79 | **0.82** | **0.74** |
| | $\delta = 25$ | 0.83 | 0.75 | **0.80** | 0.78 | 0.72 |
| RIM | $\delta = 5$ | 0.87 | 0.81 | 0.76 | 0.76 | 0.71 |
| | $\delta = 15$ | **0.91** | **0.89** | **0.86** | **0.83** | 0.73 |
| | $\delta = 25$ | 0.81 | 0.78 | 0.84 | 0.81 | **0.75** |

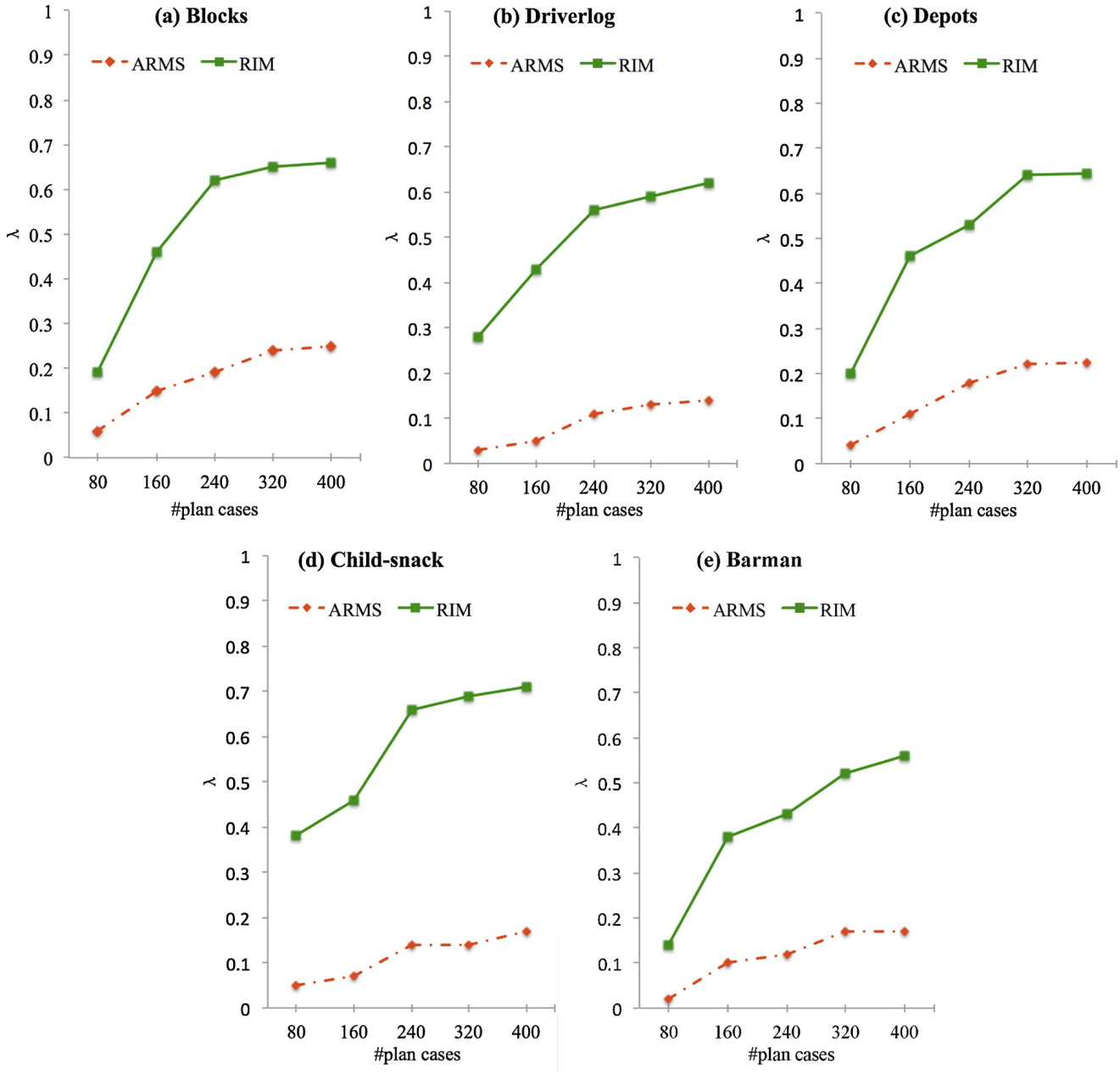


**Fig. 7.** Comparison between ARMS and RIM.

By observing all five domains in Fig. 6, we found that ML-CBP and RIM perform much better when the percentage was smaller. This indicates that exploiting multiple plan fragments, as ML-CBP and RIM do, plays a more important role when the percentage is smaller. OAKPlan does not consider this factor, i.e., it still retrieves only one case.

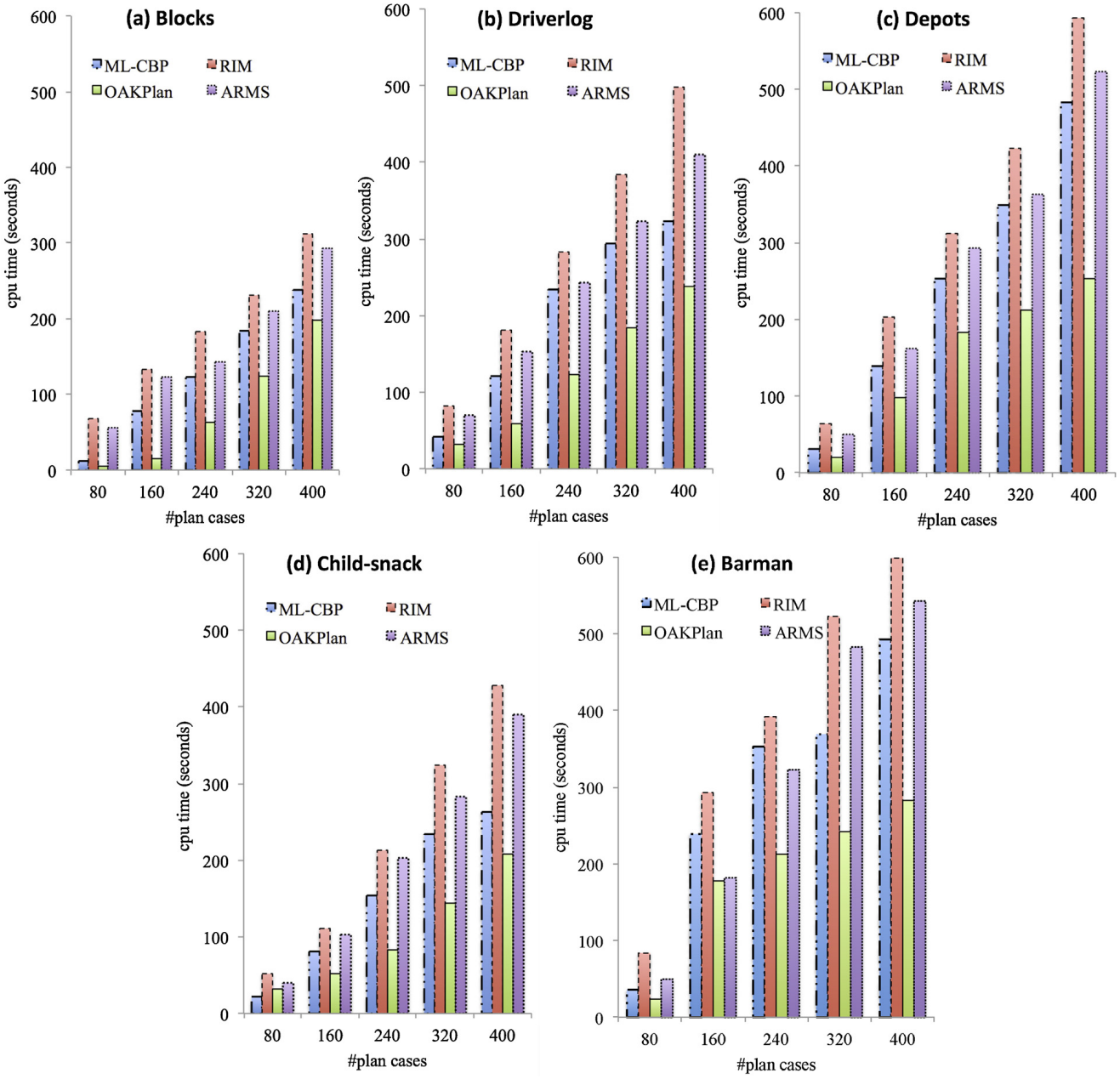


**Fig. 8.** The running time of ML-CBP, RIM, OAKPlan, ARMS.

### 6.4. *Impact of thresholds in* ML-CBP *and* RIM

We tested different support thresholds to see how they affected the accuracy. We set the completeness to be 60% and the number of plan cases to be 200. The result is shown in Table 2. The bold parts indicate the highest accuracies. We found that the threshold could not be too high or too low, as was shown in domains *blocks* and *driverlog*. A high threshold may incur *false negative*, i.e., "good" plan fragments are excluded when mining frequent plan fragments in step 3 of Algorithm 4. In contrast, a low threshold may incur *false positive*, i.e., "bad" plan fragments are introduced. Both of these two cases may reduce the accuracy. We can see that the best choice for the threshold could be 15 (the accuracies of $\delta = 15$ and $\delta = 25$ are close in *depots* and *Barman*).

### 6.5. *Impact of the learnt macro-operators in* RIM

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 100 new planning problems using the learnt models; we then learn action models using ARMS and solve the same 100 planning problems with the learnt action models. Note that since ARMS

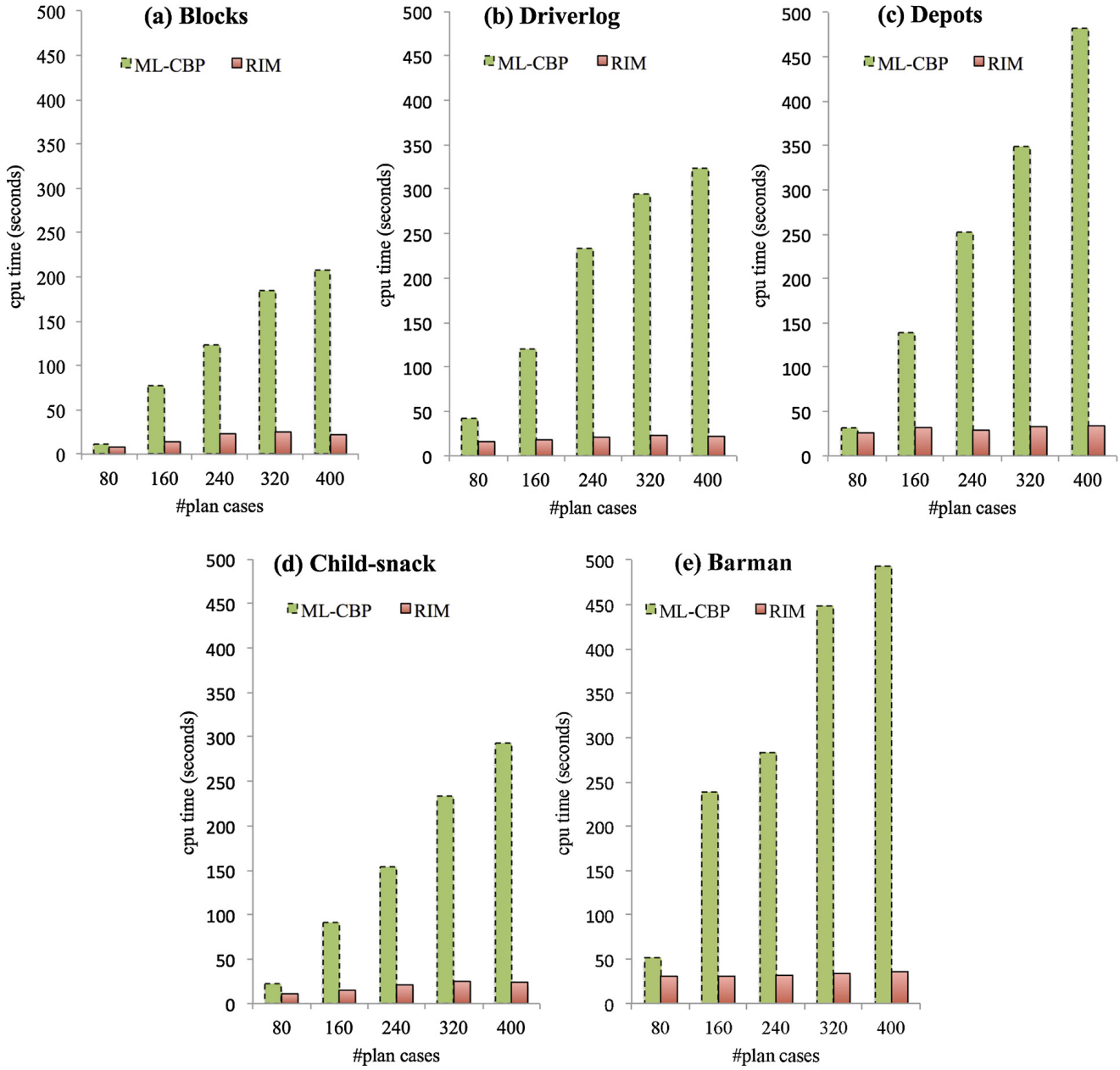


**Fig. 9.** The running time of ML-CBP and the "planning" time of RIM.

constructs action models from only plan cases, 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 $\delta$ was set to be 15 and the completeness was set to be 60%.

Fig. 7 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 cases increases, the accuracies are also getting higher. This is consistent with our intuition, since the more plan cases 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 240 plan cases for learning in the *blocks* domain. The average length of solutions is 19 when using action models learnt by ARMS; while the average length of solutions (to the same problems as solved by ARMS) is 22 when using preferences of macro-operators learnt by RIM. This is reasonable given that the plans with macro-operators have higher quality.

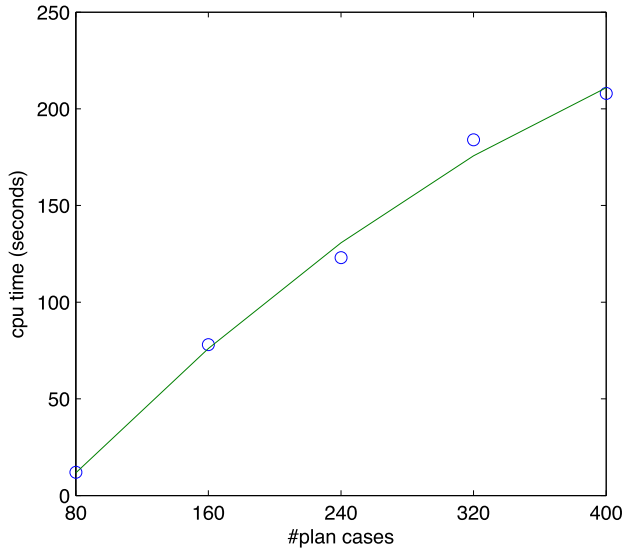


**Fig. 10.** The fit polynomial of running time of ML-CBP for blocks.

### 6.6. Running time

We show the average CPU time of ML-CBP, RIM, OAKPlan and ARMS over 100 planning problems with respect to different number of plan cases in Fig. 8. We set the completeness of domain models to be 60% and the threshold to be 15. The running time of RIM shown in Fig. 8 includes both phases, i.e., the learning and planning phases. As expected, the running time for ML-CBP, RIM, OAKPlan and ARMS increase as the number of input plan cases increase. Moreover, the total running time of RIM is larger than ML-CBP in all five domains. We note that this time includes the time RIM takes to refine the model, as well as the time it takes to use the model in solving problems. The running time of ARMS is larger than ML-CBP and less than RIM. This is because RIM takes more running time to learn additional macro operators compared to ARMS that just learns action models. Similar to RIM, compared to ML-CBP, ARMS needs more running time to refine action models. In addition, we can see that OAKPlan takes the least running time of all four algorithms. This is because OAKPlan does not spend running time to integrate multiple plan fragments or refine domain models, compared to other three algorithms.

One important difference between ML-CBP and RIM is that the model-learning phase of RIM is done just once, and the resulting model is used for solving any number of new problems. In contrast, ML-CBP doesn't have any setup costs and thus its run time is proportional to the number of new problems to be solved. To put this difference in clearer perspective, we also recorded time cost of the planning phase of RIM, assuming the learning phase was done once and applied for solving all 100 planning problems. An average of the time cost is shown in Fig. 9. Note that the running time of ML-CBP is the same as Fig. 8. From Fig. 9 we can see that time cost of the planning phase in RIM is very low comparing to ML-CBP. This is because the knowledge (the model of macro-operators and actions) needed for planning has been done by the learning phase. These results suggest that RIM can win over ML-CBP in terms of runtime, as the number of new problems to be solved increases.

As can be seen from the figure, the running time (of both ML-CBP and RIM) increases polynomially with respect to the number of input plan cases. This can be verified by fitting the relation between the number of plan cases and the running time to a performance curve with a polynomial of order 2 or 3. For example, the fit polynomial of ML-CBP for *blocks* is $-0.0008x^2 + 0.9868x - 62.4000$, whose curve is shown in Fig. 10.

## 7. Conclusion

In this paper, we considered the problem of generating robust and accurate plans, when the agent only has access to incomplete domain models, supplanted by a set of successful plan cases. We presented two novel approaches for the problem. ML-CBP is a case-based approach that leverages the incomplete model and the plan cases to solve a new problem directly by affecting case-level transfer. RIM is a model-based approach that uses the incomplete model and the plan cases to first learn a more complete model. This model contains both primitive actions as well as macro-operators that are derived from the plan cases. The learned model is then used in conjunction with an off-the-shelf planner to solve new problems. We argued that both ML-CBP and RIM go beyond existing case-based planning approaches, and action model learning approaches. We presented a comprehensive evaluation of the two approaches, both to characterize their relative tradeoffs, and to quantify their advances over existing approaches. For the former, we focused on how the number of available cases and the degree of incompleteness of the input model affect the relative performance of ML-CBP and RIM. For the latter, we compared ML-CBP to OAKPlan, an off-the-shelf state of the art case-based planner, and RIM to ARMS, an off-the-shelf

action model learning system. We believe that the insights from this work would be valuable to the planning community, given the increasing interest in supporting planning in scenarios where complete models are not available.

## Acknowledgements

We thank Dr. Tuan Nguyen for the discussion on comparing case-based and model-based approaches. Hankz Hankui Zhuo thanks the National Key Research and Development Program of China (2016YFB0201900), National Natural Science Foundation of China (U1611262), Pearl River Science and Technology New Star of Guangzhou, and Guangdong Province Key Laboratory of Big Data Analysis and Processing for the support of this research. Kambhampati's research is supported in part by the ONR grants N00014-16-1-2892, N00014-13-1-0176, N00014-13-1-0519 and N00014-15-1-2027.

## References

[1] R. Alterman, An adaptive planner, in: Proceedings of AAAI, 1986, pp. 65–71.
[2] E. Amir, Learning partially observable deterministic action models, in: Proceedings of IJCAI, 2005, pp. 1433–1439.
[3] F. Bacchus, Q. Yang, The downward refinement property, in: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, 1991, pp. 286–292.
[4] R. Bergmann, W. Wilke, Building and refining abstract planning cases by change of representation language, J. Artif. Intell. Res. 3 (1995) 53–118.
[5] P. Bertoli, M. Pistore, P. Traverso, Automated composition of web services via planning in asynchronous domains, Artif. Intell. 174 (3–4) (2010) 316–361.
[6] J. Blythe, E. Deelman, Y. Gil, Automatically composedworkflows for grid environments, IEEE Intell. Syst. 19 (4) (2004) 16–23.
[7] B. Bonet, H. Geffner, Planning with incomplete information as heuristic search in belief space, in: Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14–17, 2000, 2000, pp. 52–61.
[8] D. Borrajo, A. Roubíčková, I. Serina, Progress in case-based planning, ACM Comput. Surv. 47 (2) (Jan. 2015) 35:1–35:39.
[9] A. Botea, M. Enzenberger, M. Muller, J. Schaeffer, Macro-ff: improving AI planning with automatically learned macro-operators, J. Artif. Intell. Res. 24 (2005) 581–621.
[10] D. Bryce, C. Weber, Planning and acting in incomplete domains, in: Proceedings of ICAPS, 2011.
[11] J.G. Carbonell, Learning by analogy: formulating and generalizing plans from past experience, in: R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Eds.), Machine Learning: An Artificial Intelligence Approach, Springer, Berlin, Heidelberg, 1984, pp. 137–161.
[12] A. Coles, A. Smith, Marvin: a heuristic search planner with online macro-action learning, J. Artif. Intell. Res. 28 (2007) 119–156.
[13] S. Cresswell, T.L. McCluskey, M.M. West, Acquisition of object-centred domain models from planning examples, in: Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling, ICAPS'09, 2009.
[14] O. Etzioni, S. Hanks, D.S. Weld, D. Draper, N. Lesh, M. Williamson, An approach to planning with incomplete information, in: Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning, KR'92, Cambridge, MA, October 25–29, 1992, 1992, pp. 115–125.
[15] C. Fawcett, M. Vallati, F. Hutter, J. Hoffmann, H.H. Hoos, K. Leyton-Brown, Improved features for runtime prediction of domain-independent planners, in: Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21–26, 2014, 2014.
[16] R.E. Fikes, P.E. Hart, N.J. Niisson, Learning and executing generalized robot plans, Artif. Intell. 3 (1972) 251–288.
[17] A. Gerevini, I. Serina, Fast plan adaptation through planning graphs: local and systematic search techniques, in: Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14–17, 2000, 2000, pp. 112–121.
[18] P. Gregory, S. Cresswell, Domain model acquisition in the presence of static relations in the LOP system, in: Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7–11, 2015, 2015, pp. 97–105.
[19] N. Gupta, D.S. Nau, On the complexity of blocks-world planning, Artif. Intell. 56 (2–3) (1992) 223–254, http://dx.doi.org/10.1016/0004-3702(92)90028-V.
[20] K.J. Hammond, Case-Based Planning: Viewing Planning as a Memory Task, Academic Press, San Diego, CA, 1989.
[21] S. Hanks, D.S. Weld, A domain-independent algorithm for plan adaptation, J. Artif. Intell. Res. 2 (1995) 319–360, http://dx.doi.org/10.1613/jair.79.
[22] R. He, E. Brunskill, E. Brunskill, Puma: planning under uncertainty with macro-actions, in: Proceedings of AAAI, 2010, pp. 1089–1095.
[23] J. Hoffmann, P. Bertoli, M. Pistore, Web service composition as planning, revisited: in between background theories and initial state uncertainty, in: Proceedings of AAAI, 2007.
[24] J. Hoffmann, B. Nebel, The FF planning system: fast plan generation through heuristic search, J. Artif. Intell. Res. 14 (2001) 253–302.
[25] J. Hoffmann, B. Nebel, What makes the difference between HSP and FF?, in: Proceedings IJCAI-01 Workshop on Empirical Methods in Artificial Intelligence, 2001.
[26] G.A. Iba, A heuristic approach to the discovery of macro-operators, Mach. Learn. 3 (1989) 285–317.
[27] L.H. Ihrig, S. Kambhampati, Derivation replay for partial-order planning, in: Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31–August 4, 1994, vol. 2, 1994, pp. 992–997.
[28] L.H. Ihrig, S. Kambhampati, Storing and indexing plan derivations through explanation-based analysis of retrieval failures, J. Artif. Intell. Res. 7 (1997) 161–198.
[29] S. Kambhampati, Model-lite planning for the web age masses: the challenges of planning with incomplete and evolving domain theories, in: Proceedings of AAAI, 2007.
[30] S. Kambhampati, J.A. Hendler, A validation-structure-based theory of plan modification and reuse, Artif. Intell. 55 (1992) 193–258.
[31] R.E. Korf, Macro-operators: a weak method for learning, Artif. Intell. 26 (1985) 35–77.
[32] C.M. Li, F. Manya, J. Planes, New inference rules for Max-SAT, J. Artif. Intell. Res. 30 (October 2007) 321–359.
[33] E. Melis, J. Whittle, Internal analogy in theorem proving, in: Automated Deduction – CADE-13, 13th International Conference on Automated Deduction, Proceedings, New Brunswick, NJ, USA, July 30–August 3, 1996, 1996, pp. 92–105.
[34] D. Morwood, D. Bryce, Evaluating temporal plans in incomplete domains, in: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22–26, 2012, Toronto, Ontario, Canada, 2012.
[35] H. Muñoz-Avila, Case-base maintenance by integrating case-index revision and case-retention policies in a derivational replay framework, Comput. Intell. 17 (2) (2001) 280–294.
[36] H. Muñoz-Avila, D.W. Aha, L. Breslow, D.S. Nau, HICAP: an interactive case-based planning architecture and its application to noncombatant evacuation operations, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18–22, 1999, Orlando, Florida, USA, 1999, pp. 870–875.
[37] H. Muñoz-Avila, M.T. Cox, Case-based plan adaptation: an analysis and review, IEEE Intell. Syst. 23 (4) (2008) 75–81, http://dx.doi.org/10.1109/MIS.2008.59.
[38] M.H. Newton, J. Levine, M. Fox, D. Long, Learning macro-actions for arbitrary planners and domains, in: Proceedings of ICAPS, 2007, pp. 256–263.

[39] T.A. Nguyen, S. Kambhampati, M.B. Do, Assessing and generating robust plans with partial domain models, in: ICAPS Workshop on Planning Under Uncertainty, 2010.
[40] M. Oglietti, Understanding planning with incomplete information and sensing, Artif. Intell. 164 (1–2) (2005) 171–208.
[41] D. Olawsky, M. Gini, Deferred planning and sensor use, in: DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, Morgan Kaufmann, 1990.
[42] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, M.-C. Hsu, Mining sequential patterns by pattern-growth: the prefixspan approach, IEEE Trans. Knowl. Data Eng. 16 (11) (2004) 1424–1440.
[43] R.P.A. Petrick, F. Bacchus, A knowledge-based approach to planning with incomplete information and sensing, in: Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, April 23–27, 2002, Toulouse, France, 2002, pp. 212–222.
[44] I. Serina, Kernel functions for case-based planning, Artif. Intell. 174 (16–17) (2010) 1369–1406.
[45] L. Spalazzi, A survey on case-based planning, Artif. Intell. Rev. 16 (1) (2001) 3–36.
[46] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, Cambridge, Massachusetts, 1998.
[47] A. Torreño, E. Onaindia, O. Sapena, An approach to multi-agent planning with incomplete information, CoRR, arXiv:1501.07256, 2015.
[48] M. Vallati, I. Serina, A. Saetti, A.E. Gerevini, Identifying and exploiting features for effective plan retrieval in case-based planning, in: Proceedings of ICAPS-15, 2015.
[49] R. van der Krogt, M. de Weerdt, Plan repair as an extension of planning, in: Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling, ICAPS 2005, June 5–10, 2005, Monterey, California, USA, 2005, pp. 161–170.
[50] M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, J. Blythe, Integrating planning and learning: the prodigy architecture, J. Exp. Theor. Artif. Intell. 7 (1) (1995).
[51] T.J. Walsh, M.L. Littman, Efficient learning of action schemas and web-service descriptions, in: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI-08, 2008, pp. 714–719.
[52] Q. Yang, K. Wu, Y. Jiang, Learning action models from plan examples using weighted MAX-SAT, Artif. Intell. 171 (February 2007) 107–143.
[53] M.J. Zaki, Spade: an efficient algorithm for mining frequent sequences, Mach. Learn. 42 (2001) 31–60.
[54] L.S. Zettlemoyer, H.M. Pasula, L.P. Kaelbling, Learning planning rules in noisy stochastic worlds, in: Proceedings of AAAI, 2005.
[55] H.H. Zhuo, Crowdsourced action-model acquisition for planning, in: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25–30, 2015, Austin, Texas, USA, 2015, pp. 3439–3446.
[56] H.H. Zhuo, H. Muñoz-Avila, Q. Yang, Learning hierarchical task network domains from partially observed plan traces, Artif. Intell. 212 (2014) 134–157.
[57] H.H. Zhuo, T.A. Nguyen, S. Kambhampati, Model-lite case-based planning, in: Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14–18, 2013, Bellevue, Washington, USA, 2013.
[58] H.H. Zhuo, T.A. Nguyen, S. Kambhampati, Refining incomplete planning domain models through plan traces, in: Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3–9, 2013, 2013, pp. 2451–2458.
[59] H.H. Zhuo, Q. Yang, Action-model acquisition for planning via transfer learning, Artif. Intell. 212 (2014) 80–103.
[60] H.H. Zhuo, Q. Yang, D.H. Hu, L. Li, Learning complex action models with quantifiers and logical implications, Artif. Intell. 174 (18) (2010) 1540–1569.