Model-Lite Case-Based Planning

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

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 a novel solution for this challenge that combines generative planning on incomplete domain models with a library of plan cases that are known to be correct. While this was arguably the original motivation for case-based planning, most existing case-based planners assume (and depend on) from-scratch planners that work on complete domain models. In contrast, our approach views the plan generated with respect to the incomplete model as a "skeletal plan" and augments it with directed mining of plan fragments from library cases. We will present the details of our approach and present an empirical evaluation of our method in comparison to a state-of-the-art case-based planner that depends on complete domain models.

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, Pistore, and Traverso 2010; Hoffmann, Bertoli, and Pistore 2007)) and work-flow management (c.f. (Blythe, Deelman, and Gil 2004)) 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 (c.f. (Kambhampati 2007)) 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. 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. This is only possible if the planner has access to additional sources of knowledge besides the incomplete domain model.

Interestingly, one of the original motivations for case-based planning was also the realization that in many domains complete domain models are not available, but successful plans (cases) are. Over years however, case-based planning systems deviated from this motivation and focused instead on "plan reuse" where the motivation is to improve the performance of a planner operating with a complete domain model. In this paper, we return to the original motivation by considering "model-lite case-based planning." In particular, we consider plan synthesis when the planner has an incomplete domain theory, but has access to a library of plans that "worked" in the past. This plan library can thus be seen as providing additional knowledge of the domain over and above the incomplete domain theory.

Our task is to effectively bring to bear this additional knowledge on plan synthesis to improve the correctness of the plans generated. 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 will describe the details of our framework, called `ML-CBP` and present a systematic empirical evaluation of its effectiveness. We compare the effectiveness of our model-lite case-based planner with OAKPlan (Serina 2010), the current state-of-the-art model-complete case-based planner.

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 `ML-CBP` algorithm. Finally, we evaluate `ML-CBP` in three planning domains, and compare its performance to OAKPlan.

Related Work

As the title implies, our work is related both to case-based planning and model-lite planning. As mentioned in the introduction, our work is most similar to the spirit of original case-based planning systems such as CHEF (Hammond 1989) and PLEXUS (Alterman 1986), which 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. Our work however differs from CHEF in two ways. First, unlike us, CHEF assumes access to a (more)

complete domain model during its debugging stage. Second, CHEF 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. The post-CHEF case-based planning work largely focused on having access to a from-scratch planner operating on complete domain models (c.f. (Kambhampati and Hendler 1992; Veloso et al. 1995)). The most recent of this line of work is OAKPlan (Serina 2010), which we compare against.

The recent focus on planning with incomplete domain models originated with the work on "model-lite planning" (Kambhampati 2007). 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. (Nguyen, Kambhampati, and Do 2010) and Weber et. al. (Bryce and Weber 2011) 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.

A third strand of research that is also related to our work is that of action model learning. Work such as (Yang, Wu, and Jiang 2007; Zhuo et al. 2010; Zettlemoyer, Pasula, and Kaelbling 2005; Walsh and Littman 2008; Cresswell, McCluskey, and West 2009) focuses on learning action models directly from observed (or pre-specified) plan traces. 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. Finally, in contrast, work such as (Amir 2005), as well as much of the reinforcement learning work (Sutton and Barto 1998) 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 (c.f. (Ihrig and Kambhampati 1997)).

Problem Definition

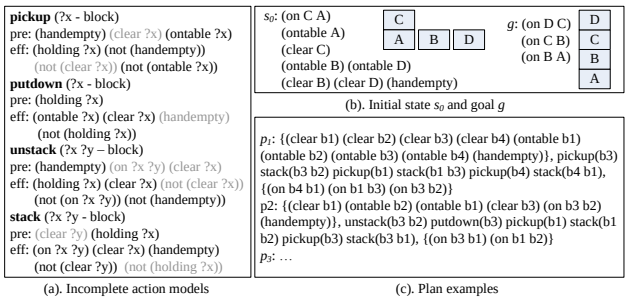

Figure 1: An input example of ML−CBP for domain *blocks*.

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 $\mathcal{R}_\mathcal{O}$ 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.

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, 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 no action model in $\mathcal{A}$ is missing, 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 $\widetilde{\mathcal{P}} = \langle \mathcal{O}, s_o, g \rangle$ 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 $\mathcal{M}^*$.

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 (Bacchus and Yang 1991) does not hold with model incompleteness).

An example input of our planning problem in *blocks*[2] domain is shown in Figure 1. 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 Figure 1(a), the dark parts indicate the missing predicates. In Figure 1(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)".

The ML−CBP Algorithm

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.

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

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

Algorithm 1 The ML−CBP algorithm

Input: $\widetilde{\mathcal{P}} : \langle \mathcal{O}, s_0, g \rangle$, $\widetilde{\mathcal{M}}$, 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 φ:
φ=*build_fragments*($\widetilde{\mathcal{P}}$,$\mathcal{C}$);
3: mine a set of frequent plan fragments $\mathcal{F}$:
$\mathcal{F}$=*freq_mining*(φ);
4: p^{sol}= *concat_frag*($\mathcal{L}$, $\mathcal{F}$);
5: **return** p^{sol};

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

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: $\mathcal{L} = \emptyset$;
2: **for** each proposition $f \in g$ **do**
3: generate a plan with $\widetilde{\mathcal{M}}$, denoted by a set of causal pairs $\mathcal{L}'$, to transit s_0 to f;
4: $\mathcal{L} = \mathcal{L} \cup \mathcal{L}'$;
5: **end for**
6: **return** $\mathcal{L}$;

step 3 of 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.

Example 1: *As an example, causal pairs generated for the planning problem given in Figure 1 is* $\{\langle$ *pickup(B), stack(B A)* $\rangle$, $\langle$*unstack(C A), stack(C B)*$\rangle$, $\langle$ *pickup(D), stack(D C)* $\rangle\}$.

Creating Plan Fragments

In the procedure *build_fragments* of Algorithm 1, we would like to build a set of plan fragments φ 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 example 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 Figure 1, 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.

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 φ 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 Figure 1,* 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.*

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 (Zaki 2001; Pei et al. 2004) 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 δ 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 δ and use the SPADE algorithm (Zaki 2001) 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 δ 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)*}.

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*.

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*, which is indicated by common action subsequence. 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 Figure 2. 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.

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** $\mathcal{L} \neq \emptyset$ **do**
2: randomly select a pair $\langle a_i, a_j \rangle \in \mathcal{L}$;
3: randomly select some $f \in \mathcal{F}$, such that: $(a_i \in f \vee a_j \in f)$ and *share*(p^{sol}, f) =**true**;
4: **if** there is no such f, **return** $\langle\rangle$;
5: p^{sol}=*append*(p^{sol}, f);
6: $\mathcal{L}$=*removelinks*(p^{sol}, $\mathcal{L}$);
7: **end while**
8: **return** p^{sol};

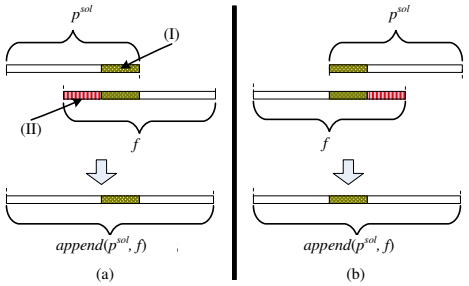

Figure 2: (a). f is concatenated at the end of p^{sol}; (b). f is concatenated at the beginning of p^{sol}; Part (I) is the maximal action subsequence; Part (II) is the action subsequence that is different from p^{sol}.

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 $\mathcal{L}$ is satisfied and will be removed according to step 6 of Algorithm 3. The result is shown in the fourth row.

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. 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.

Experiments

Dataset and Criterion

We evaluate our ML−CBP algorithm in three planning domains: *blocks*[2], *driverlog*[3] and *depots*[3]. In each domain, we generate from 40 to 200 plan cases using a classical planner such as FF planner[4] 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 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 examples are completely different from those used in testing problems; thus plan cases and new problems don't share common propositions.

We measure the accuracy of our ML−CBP algorithm as the percentage of correctly solved planning problems. Specifically, we exploit ML−CBP 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 λ, 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, we assume that we have *complete* domain models available for executing generated solutions.

Experimental Results

We evaluated ML−CBP in the following aspects: (1) the comparison between ML−CBP and OAKPlan; (2) the change of accuracies with respect to different support threshold δ; (3) the average of plan lengths; (4) the running time of ML−CBP.

Comparison between ML−CBP and OAKPlan We compared ML−CBP with the state-of-the-art case-based planning system OAKPlan (Serina 2010). Both OAKPlan and ML−CBP 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 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 would like to first test the change of the accuracy when the number of plan examples increases. We set the percentage of completeness as 60%, and the threshold δ as 15. We varied the number of plan cases from 40 to 200 and run ML−CBP to solve 100 planning problems. Figure 3 shows the accuracy λ with respect to the number of plan cases used.

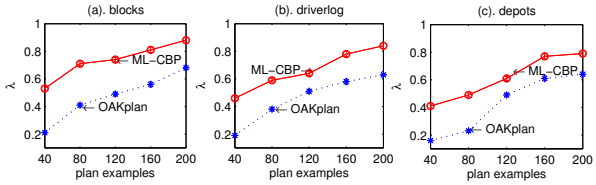

Figure 3: Accuracy w.r.t. number of plan cases.

From Figure 3, we found that both accuracies of ML−CBP and OAKPlan generally became larger when the number of plan examples increased. This is consistent with our intuition, since there is more knowledge to be used when plan examples become larger. On the other hand, we also found that ML−CBP generally had higher accuracy than OAKPlan in all the three domains. This is because ML−CBP exploits the information of incomplete models to mine *multiple* high quality plan fragments, i.e., ML−CBP integrates the knowledge from both incomplete models and plan examples, 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. By observation, we found that the accuracy of ML−CBP was no less than 0.8 when the number of plan examples was more than 160.

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 Figure 4.

We found that both accuracies of ML−CBP and OAKPlan increased when the percentage of completeness increased, due to more information provided when the percentage increasing. When the percentage is 100%, both ML−CBP and OAKPlan can solve all the solvable planning problems successfully. Similar to Figure 3, ML−CBP functions better than

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

[4]http://members.deri.at/∼joergh/ff.html

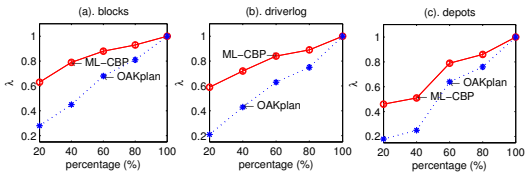

Figure 4: Accuracy w.r.t. percentage of completeness.

OAKPlan. The reason is similar to Figure 3, i.e., simultaneously exploiting both knowledge from incomplete domain models and plan cases could be helpful.

By observing all three domains in Figure 4, we found that ML−CBP functioned much better when the percentage was smaller. This indicates that exploiting multiple plan fragments, as ML−CBP does, plays a more important role when the percentage is smaller. OAKPlan does not consider this factor, i.e., it still retrieves only one case.

Average of plan length We calculated an average of plan length for all problems successfully solved by ML−CBP when δ was 15, the percentage of completeness was 60%, and 200 plan examples were used. As a baseline, we exploited FF to solve the same problems using the corresponding complete domain models and calculate an average of their plan length. The result is shown in Table 1.

Table 1: Average of plan length

domains	blocks	driverlog	depots
ML−CBP	46.8	83.4	**95.3**
FF	**35.2**	**79.2**	96.7

From Table 1, we found that the plan length of ML−CBP was larger than FF in some cases, such as *blocks* and *driverlog*. However, it was also possible that ML−CBP had shorter plans than FF (e.g., *depots*), since high quality plan fragments could help acquire shorter plans.

Varying the support threshold We tested different support thresholds to see how they affected the accuracy. We set the completeness to be 60%. 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 1. 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*).

Table 2: Accuracy with respect to different thresholds.

threshold	blocks	driverlog	depots
$\delta = 5$	0.80	0.78	0.73
$\delta = 15$	**0.88**	**0.84**	0.79
$\delta = 25$	0.83	0.75	**0.80**

The running time We show the average CPU time of our ML−CBP algorithm over 100 planning problems with respect to different number of plan cases 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 cases and the running time to a performance curve with a polynomial of order 2 or 3. For example, the fit polynomial for *blocks* is $-0.0022x^2 + 1.1007x - 45.2000$.

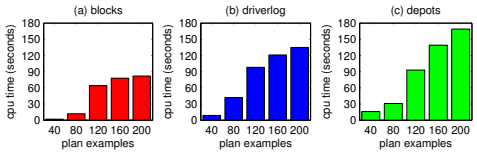

Figure 5: The running time of our ML−CBP algorithm

Conclusion

In this paper, we presented a system called ML−CBP for doing model-lite case-based planning. ML−CBP is able to integrate knowledge from both incomplete domain models and a library of plan examples to produce solutions to new planning problems. Our experiments show that ML−CBP is effective in three benchmark domains compared to case-based planners that rely on complete domain models. Our approach is thus well suited for scenarios where the planner is limited to incomplete models of the domain, but does have access to a library of plans correct with respect to the complete (but unknown) domain theory. Our work can be seen as a contribution both to model-lite planning, which is interested in plan synthesis under incomplete domain models, and the original vision of case-based planning, which aimed to use a library of cases as an extensional representation of planning knowledge.

While we focused on the accuracy of the generated plans rather than on the efficiency issues in the current paper, we do believe that efficient mapping techniques such as those used in OAKPlan can certainly be integrated into ML-CBP; we are currently investigating this hypothesis. While we used cases to do case-based planning directly, an alternative approach is to use them to improve the incomplete model. In a parallel effort (Zhuo, Nguyen, and Kambhampati 2013), we have indeed developed an approach called RIM that extends the ARMS family of methods for learning models from plan traces (Zhuo et al. 2010; Yang, Wu, and Jiang 2007) so that they can benefit from both the incomplete model, and the macro-operators extracted from the plan traces. In future, we hope to investigate the relative tradeoffs between these different ways of exploiting the case knowledge.

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 Guangzhou Science and Technology Project (No. 2011J4300039) 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, R. 1986. An adaptive planner. In *Proceedings of AAAI*, 65–71.

Amir, E. 2005. Learning partially observable deterministic action models. In *Proceedings of IJCAI*, 1433–1439.

Bacchus, F., and Yang, Q. 1991. The downward refinement property. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 286–292.

Bertoli, P.; Pistore, M.; and Traverso, P. 2010. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence Journal* 174(3-4):316–361.

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

Bryce, D., and Weber, C. 2011. Planning and acting in incomplete domains. In *Proceedings of ICAPS*.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2009. Acquisition of object-centred domain models from planning examples. In *Proceedings of ICAPS-09*.

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

Hoffmann, J.; Bertoli, P.; and Pistore, M. 2007. Web service composition as planning, revisited: In between background theoriesandinitial state uncertainty. In *Proceedings of AAAI*.

Ihrig, L. H., and Kambhampati, S. 1997. Storing and indexing plan derivations through explanation-based analysis of retrieval failures. *Journal of Artificial Intelligence Research* 7:161–198.

Kambhampati, S., and Hendler, J. A. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence Journal* 55:193C258.

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

Nguyen, T. A.; Kambhampati, S.; and Do, M. B. 2010. Assessing and generating robust plans with partial domain models. In *ICAPS Workshop on Planning under Uncertainty*.

Pei, J.; Han, J.; Mortazavi-Asl, B.; Wang, J.; Pinto, H.; Chen, Q.; Dayal, U.; and Hsu, M.-C. 2004. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering* 16(11):1424–1440.

Serina, I. 2010. Kernel functions for case-based planning. *Artificial Intelligence* 174(16-17):1369–1406.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: MIT Press.

Veloso, M.; Carbonell, J.; Prez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence* 7(1).

Walsh, T. J., and Littman, M. L. 2008. Efficient learning of action schemas and web-service descriptions. In *Proceedings of AAAI-08*, 714–719.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence Journal* 171:107–143.

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

Zettlemoyer, L. S.; Pasula, H. M.; and Kaelbling, L. P. 2005. Learning planning rules in noisy stochastic worlds. In *Proceedings of AAAI*.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and implications. *Artificial Intelligence* 174(18):1540 – 1569.

Zhuo, H. H.; Nguyen, T.; and Kambhampati, S. 2013. Refining incomplete planning domain models through plan traces. In *Proceedings of IJCAI*.