Solving Large-Scale Planning Problems by Decomposition and Macro Generation

Masataro Asai and **Alex Fukunaga**

Abstract

Large-scale classical planning problems such as factory assembly problems pose a significant challenge for domain-independent planners. We propose a macro generation method which automatically identifies subcomponents of the problem that can be solved independently. The decomposition is based solely on the instance itself, with no need of training instances. We show experimentally that our approach can be used to solve large problem instances that are beyond the reach of current state-of-the-art satisficing planners.

Introduction

Although domain-independent planners have made significant strides in recent years, large-scale problems continue to pose a challenge due to the fundamental computational complexity of planning. One approach to scaling the performance of domain-independent planners is to exploit specific types of problem structure that are naturally present in many domains. In this paper, we propose a method for exploiting decomposable structures that are present in assembly and transportation type domains, where a large problem can be decomposed into subtasks that can be solved (mostly) independently. We focus on finding satisficing plans in classical planning, where the objective is to find (possibly suboptimal) plans that achieve the goals.

Consider a factory assembly problem where the objective is to assemble 20 instances of Widget A and 20 instances of Widget B. Even in a relatively simple version of this domain, it has been shown that state-of-the-art planners struggle to assemble 4-6 instances of a single product (Asai and Fukunaga 2014). However, it is clear that this problem consists of serializable subgoals (Korf 1987). The obvious strategy is to first find a plan to assemble 1 instance of Widget A and a plan to assemble 1 instance of Widget B, and then combine these building blocks in order to assemble 20 instances of both widgets.

In principle, such serializable problems *should* be easy for modern planners equipped with powerful heuristic functions and other search-enhancing techniques. In fact, techniques such as *probes* (Lipovetzky and Geffner 2011) explicitly target the exploitation of serializable subgoals and have been shown to be quite effective. However, as we show in our experiments, there is a limit to how far current, state-of-the-art planners can scale when faced with very large, serializable problems. The ability to assemble 4-6 instances each of Widgets A and B does not necessarily imply the ability to assemble 20 instances each of Widgets A and B.

In this paper, we propose an approach to solving such classes of planning problems by automatic decomposition into subproblems, **C**omponent **A**bstraction **P**lanner (CAP). The subproblems are solved using standard, domain-independent planners, and solutions to these subproblems are converted to macro operators and added to the original domain, and this enhanced domain is solved again by a standard, domain-independent planner. For example, in the assembly domain mentioned above, our system extracts a set of subproblems. Each subproblem corresponds to the assembly of a single product, like Widget A and Widget B.

The notable diffirence of CAP from all existing macro-based methods to our knowledge is that CAP macros do not aim at *reusable* (highly applicable) plan fragments. Most of the previous work, e.g., Macro-FF/SOL-EP/CA-ED(Botea et al. 2005), Wizard(Newton et al. 2007) or MUM(Chrpa, Vallati, and McCluskey 2014), target a learning scenario such as IPC2011 Learning Track, where the system first learns domain control knowledge (DCK) from the training instances, and later use this DCK to solve the different, testing instances. Due to this setting, the macros should encapsulate the knowledge *reusable* across the problems in a domain. Marvin (Coles and Smith 2007) learns macros in a satisficing track setting, but it also seek macros reusable within the same problem instance (Coles and Smith 2007, page 10).

CAP is different from both groups. First, CAP targets a satisficing track setting which precludes transfer of DCK across problem instances. Second, even within the same instance, the primary benefit of CAP macros comes from its problem decomposition, which does not require reusability.

Consider the IPC2011 Barman domain, which requires many different cocktails to be made. Each cocktail (subproblem) has a significantly different recipe. Previous learning systems such as MUM do not find any macros for Barman (Chrpa, Vallati, and McCluskey 2014, page 8) because such recipes are not reusable across instances (the learning track settings). However, a Barman instance can be decom-

posed into easy, independent cocktail mixing problems, and our results shows that CAP significantly increases the coverage on Barman. Also, when all recipes are different, search for reusable macros *within the same problem* will fail. However, given a problem of mixing cocktail A, B and C, there is no need to mix A twice – to use the macro for A twice.

CAP seeks to decompose problems, and weave together solutions to subproblems. It does not aim at the learning track, and decomposition does not require reusability. Thus, CAP macros do not have to be reusable across instances, nor within the same instance – reusability (high applicability) is unnecessary to make CAP macros useful. Our results shows that the decomposition itself is highly effective.

The second biggest difference between CAP macros and others is its capability of finding *large number of informative, very long macros*. It doesn't require filtering methods, nor upper bound in its length, while the branching factor is still suppressed. In contrast, all of MacroFF, Wizard and MUM filters their macros somehow. In terms of macro length, MacroFF/CA-ED generates short (2-steps in practice) macros (Botea, Müller, and Schaeffer 2004, page 601), and MUM finds macros "often of length of 2 or 3, occasionally (...) 5"(Chrpa, Vallati, and McCluskey 2014). Wizard has "Macro size limits"(Newton et al. 2007). An older version of Marvin "is necessary to introduce an upper bound on the plan length"(Coles and Smith 2004, page 25).

Our approach uses the notion of component abstractions, which was originally introduced by (Botea, Müller, and Schaeffer 2004). However, we take the idea of component abstraction much further: First, we identify *component tasks* consisting of the initial and goal propositions relevant to 1 component, and generate large macros that solve an entire subproblem. These *component macros* are added to the domain, and a standard domain-independent planner is used to solve the satisficing problem for the enhanced domain. As described later, these macros are all *zero-ary* and do not increase the branching factor.

CAP is targeted to the satisficing track and is not a learner. There is no need to provide additional hand-picked training instances. However, CAP have the *planner independency* in common with the traditional learners. Our system is implemented as a wrapper for any PDDL (STRIPS + action cost) based planner – a preprocessing phase performs fully automated component analysis and macro generation, resulting in an enhanced domain to be solved by a standard domain-independent planner. While the plateau-escaping macros in Marvin is closely connected to the search algorithm, CAP can be easily combined with various planners and heuristics, including FF (Hoffmann and Nebel 2001), configurations of Fast Downward (Helmert 2006), Probe (Lipovetzky and Geffner 2011) and even Marvin.

The resulting system, CAP, is capable of solving very large problems (whose decomposability is unknown prior to the search) that are beyond the range of standard, state-of-the-art planners. In our experiments, for example, CAP can solve Woodworking instances with 79 parts with 120% wood – in contrast, the largest IPC2011 instances of Woodworking has 13 parts, and state-of-the-art planner Fast Downward can solve upto 23 parts (with 120% wood). Even Probe, which can efficiently solve IPC2011 Woodworking instances with only 31 search nodes (Lipovetzky and Geffner 2011), can solve upto 46 parts with 140% wood. Our system is ready for download from https://github.com/guicho271828/CAP .

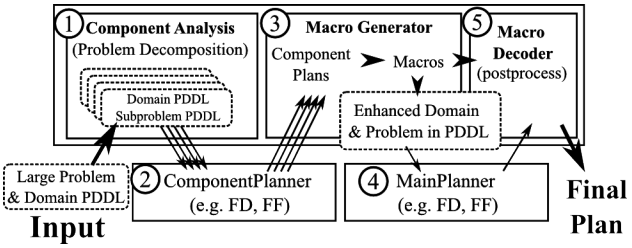

Figure 1: CAP System Overview. ComponentPlanner and MainPlanner are domain-independent planners, e.g., FD/lama (Helmert 2006), FF (Hoffmann and Nebel 2001). They can be the same planner, or different planners.

The rest of the paper is organized as follows. First, we explain our overall approach to decomposition-based solution to large-scale problems. Next we describe our method of decomposing large problems into smaller subproblems. Then we describe how we generate informative large macros, each corresponding to the result of solving each subproblem. Finally, we experimentally evaluate the effectiveness of our approach.

Overview of CAP

Given a large-scale problem such as the previously described heterogeneous cell assembly problems or Barmin problems, we propose an approach to solving them by automatically decomposing them into independent subproblems and sequencing their solutions. Figure 1 gives an overview of our overall approach, **C**omponent **A**bstraction **P**lanner (CAP). CAP performs the following steps:

1. *Component Analysis (Problem Decomposition)*: Perform a static analysis of the PDDL problem based on (Botea, Müller, and Schaeffer 2004) in order to identify the independent components. For each such component, extract the corresponding relevant portion of the initial and goal condition in the original problem, then form a subproblem called a *component problem*.
2. *ComponentPlanner*: Solve the subproblems with a domain-independent planner (ComponentPlanner) and obtain the subplans.
3. *Macro generation*: For each plan, generate a large macro operator by concatenating its actions.
4. *Main Search by MainPlanner*: Solve the original problem with a standard domain-independent planner (MainPlanner), using a new PDDL domain enhanced with macros.
5. *Decoding*: Finally, macros in the result plan are decoded into the primitive actions.

Component Abstraction, Based on (Botea, Müller, and Schaeffer 2004)

We identify the subproblems of large-scale problems using an extended version of Component Abstraction algorithm

(Botea, Müller, and Schaeffer 2004). It first builds a *static graph* of the problem and cluster it into disjoint subgraphs called *abstract components*. This algorithm requires typed PDDL, but the types can be automatically added to untyped domains using methods such as TIM (Fox and Long 1998).

The static graph of a problem Π is an undirected graph, where the nodes are the objects in the problem and the edges are the static facts in the initial state. *Static facts* are the facts (propositions) that are never added nor removed by any of the actions and only possibly appear in the preconditions. The antonym for *static* is *fluent*.

Figure 2 illustrates the static graph of a simple assembly problem with 6 objects. The planner is required to assemble 2 products a0 and a1 with parts b0 (for a0) and b1 (for a1). The fill pattern painted of each node indicates its type, e.g. b0 and b1 of type product are wave-patterned (◎). The edge (a0, b0) corresponds to a predicate specifying that part b0 is a "part-of" a0, which means it should be assembled with a0. Clearly, this kind of specification is static in the problem and is never modified by any action. Also, the edge (b0, red) indicates a static fact that part b0 "can be painted in red" (allowed to be painted red somewhere in the plan), as opposed to, e.g., "temporarily painted" red (which may be modified by some actions like "paint" or "clean").

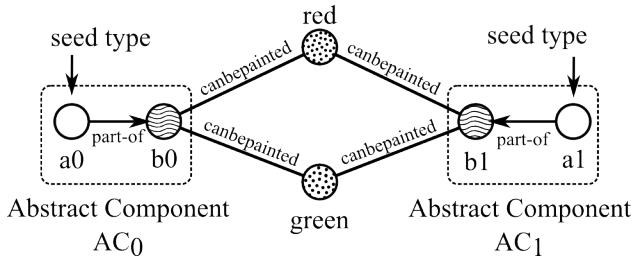

Figure 2: Example of a static graph and components.

For such a static graph, the following is a Component Abstraction algorithm based on (Botea, Müller, and Schaeffer 2004) with some modifications for our purpose. First, for each (ground) goal preposition, collect the *types* of their parameters into a queue T and initialize the final results $R = \emptyset$. Then, run the following procedure (text in brackets "[]" describes the corresponding behavior in Figure 2):

1. If T is empty, return R. Else, pop one *seed type* s from T [○ is selected as s].
2. For each object (node) o of type s, initialize single-node abstract components and store them in a set C. [The seed objects are $\{$a0, a1$\}$ and the components are $C = \{$(a0), (a1)$\}$.]
3. Select one *fringe* fact $f(a, b)$ of some component $c \in C$. A static fact $f(a, b)$ is a *fringe* of c if a belongs to the component ($a \in c$), and b does not belong to any components ($b \notin c', \forall c' \in C$). An n-ary predicate $p(x_1, \ldots x_n)$ is treated as combinations of binary predicates $p_{ij}(x_i, x_j)(j \neq i)$. [$f$ =(part-of a0 b0).] Mark the name of f ["part-of"] as *tried* and do not select it twice. If no such f exists, update $R = R \cup C$, then reset $C = \emptyset$, and return to step 1.
4. If f exists, collect all fringe facts of the same name as f. [(part-of a0 b0) and (part-of a1 b1).]
5. Extend the components by merging each with the arguments in the selected facts. [$C = \{$(a0), (a1)$\}$ are updated to $\{AC_0 =$ (a0 b0), $AC_1 =$ (a1 b1)$\}$] However, if the resulting components share any part of the structure, we discard all these new nodes. [If we extend these components further by merging ◎s, it results in (a0 b0 red green),(a1 b1 red green) which share red and green. This extension is discarded.] Finally, go back to step 3.

The resulting components form an equivalence class called *abstract type*. Components are of the same abstract type when their clustered subgraphs are isomorphic.

One difference between our procedure and that of (2004) is that we try all seed types s as long as some object of type s exist in the goals, while (2004) selects s randomly, regardless of the goal. In addition, their procedure stops and return C immediately when fully extended C meets some (heuristically chosen) criteria, while we collect C into R and use all results of the iterations.

Component Tasks and Component Plans

While (2004) generated short macros by using abstract components to identify "related" actions, we now propose a novel approach which fully exploits the structure extracted by component abstraction.

For an abstract component X, we can define a *component task* as a triple of (1) X, (2) *init* (X) – a subset of propositions from the initial states relevant to X, and (3) *goal* (X) – a subset of goal propositions relevant to X. In order to find *init* (X) and *goal* (X), we define a fluent version of fringe facts. A *fluent fringe fact* $f(a, b)$ of X is defined as follows: First of all, it should be a fluent predicate, that is, a predicate that can be added or removed by some action. Also, one of its parameters (e.g. a) belongs to X and none of the other parameters (e.g. b) belong to any other components. For each X, we collect all fluent fringe facts in the initial and goal condition for *init* (X) and *goal* (X), respectively. For example, the initial state of the problem in Figure 2 may contain a fact (not-painted b0), a fluent which takes b0 $\in AC_0$ and may be removed by some actions like (paint ?b). Similarly, there may be facts like (being b0 red), which is a goal condition specific to AC_0.

Since a component task is a compact representation of a subproblem, we can expand it into the full PDDL problem called *component problem*. Also, solving a component problem yields a *component plan*. Let $X = \{o_0, o_1 \ldots\}$ be an abstract component. The original planning problem can be expressed as 4-tuple $\Pi = \langle \mathcal{D}, O, I, G \rangle$ where $\mathcal{D}$ is a domain, O is the set of objects and I, G is the initial/goal condition. Also, let X' be another component of the same abstract type, which is written as $X \approx X'$. Then a component problem is a planning problem $\Pi_X = \langle \mathcal{D}, O_X, I_X, G_X \rangle$ where:

$$O_X = X \cup E_X, \quad E_X = \{O \ni o \mid \forall X' \approx X; o \notin X'\},$$
$$I_X = \{I \ni f \mid \textit{params}\,(f) \subseteq O_X\}, \quad G_X = \textit{goal}\,(X)$$

Informally, O_X contains X, but does not contain the siblings of X, the components of the same abstract type. E_X

means *environment objects* that are not part of any such siblings. In I_X, any ill-defined propositions (containing the removed objects) are cleaned up.

We solve this component problem Π_X with a domain-independent planner such as Fast Downward (Helmert 2006) or Fast Forward (Hoffmann and Nebel 2001) using a very short time limit (30 seconds for all experiments in this paper). This is usually far easier than the original problem due to the relaxed goal conditions. However, since Π_X is mechanically generated by removing objects and part of the initial state, Π_X may be UNSAT or ill-defined (e.g., G_X contains reference to the removed objects). If ComponentPlanner fails to solve Π_X for any reason, the component is simply ignored in the following macro generation phase.

Component Macro Operators

At this point, we have decomposed the problem into components and generated component plans which solve these components independently. However, in general, it is *not* possible to solve a problem by simply concatenating the component plans, because the decompositions (component tasks) and their corresponding component plans are based on the local reasoning on the parts of the overall problem.

Therefore, instead of trying to directly concatenate component plans, we generate macro operators called *component macros* based on the component plans, and add them to the original PDDL domain. This enhanced domain is then solved using a standard domain-independent planner (MainPlanner in Figure 1). It is intended that the planner will use component macros to solve subproblems, inserting additional actions as necessary. If the resulting plan uses macro operators, they are mapped back to corresponding ground instances of the primitive actions in the original domain. On the other hand, it is possible that some (or all) component macros are not used at all, and the planner relies heavily on the primitive actions in the original PDDL domain.

A component plan is converted to a component macro as follows: Suppose we have two actions a_1: $\langle \mathit{params}_1, \mathit{pre}_1, e_1^+, e_1^-, c_1 \rangle$ and a_2: $\langle \mathit{params}_2, \mathit{pre}_2, e_2^+, e_2^-, c_2 \rangle$, where params_i is the parameters of action a_i, pre_i are the preconditions of a_i, e_i^+ and e_i^- are the add and delete effects of a_i, and c_i is the cost of a_i. A merged action equivalent to sequentially executing a_1 and a_2 is defined as

$$a_{12} = \langle \mathit{params}_1 \cup \mathit{params}_2,\ \mathit{pre}_1 \cup (\mathit{pre}_2 \setminus e_1^+),\ (e_1^+ \setminus e_2^-) \cup e_2^+,\ (e_1^- \setminus e_2^+) \cup e_2^-,\ c_1 + c_2 \rangle \quad (1)$$

Iteratively folding this merge operation over the sequence of actions in the component plan, results in a single, ground macro operator that represents the application of the plan.

Feasibility of CAP macros, or their internal consistency, is already guaranteed as long as the ComponentPlanner is a sound planner. Note that, since our macros are *literally* the solution of the component problem (output as a PDDL problem file), all merged actions are fully grounded. Our macros are always *zero-ary*, unlike the traditional macro operators which are the combinations of several parametrized actions. This has two important effects: First, since it is not parametrized and is applicable to the limited situations, each macro does not increase the branching factor significantly. Next, it does not significantly increase the number of ground actions in the planner. Current forward-search based planners tend to instantiate ground actions from action schema, but the ground actions increase exponentially to the number of parameters. Longer parametrized macros suffers from this explosion, instantiating multitude of ground instances of such macro (the overwhelming majority of which are useless). In contrast, a zero-ary macro results in only 1 instance and does not allow the meaningless combinations of parameters.

Experimental Results

In all experiments below, planners are run on an Intel Xeon E5410@2.33GHz CPU. As baselines, we evaluated FF, Probe, and 2 configurations of Fast Downward: (a) FD/lama (iterated search), (b) FD/h_{cea} (greedy best first search on context-enhanced additive heuristic).

We tested the following CAP configurations: CAP(FF), CAP(FD/lama), CAP(Probe) and CAP(FD/h_{cea}), which use FF, FD/lama, Probe, and FD/h_{cea}, respectively as both the ComponentPlanner (for component planning) and the MainPlanner (for solving the macro-enhanced problem generated by CAP). It is possible for ComponentPlanner and MainPlanner to be different planners (Figure 1): Some planners may be better suited as the ComponentPlanner rather than the MainPlanner, and vice versa, so mixed configurations, e.g., CAP(FF +FD/lama), which uses FF as ComponentPlanner and FD/lama as MainPlanner, are possible. However, in this paper, for simplicity, we focus on configurations where ComponentPlanner and MainPlanner are the same.

Treatment of domains with action costs depend on the ComponentPlanner (CP) and MainPlanner (MP) capabilities: (1) both the CP and the MP handle action costs like CAP(FD/lama) – CAP uses action costs during all stages. (2) Neither the CP nor MP handle action costs like CAP(FF) – CAP removes action costs while planning, but in Table 1, plan costs are evaluated using the original domain definition (with costs).

Evaluation in IPC settings

We evaluate CAP using three sets of instences: **(1) IPC2011 seq-sat instances**, **(2) large instances** of IPC2011 seq-sat domains, and **(3) IPC2011 learning track, testing instances**. In these experiments, all planners are run IPC seq-sat settings i.e. $30[\text{min}], 2[\text{GB}]$ memory limits. All phases of CAP (including preprocessing, main search and postprocessing) are included in these resource limitations.

We emphasize that the **(1)** is not the main target of CAP – the motivation for CAP is scalability on **(2)** and **(3)**, which may or may not be decomposable but more difficult than **(1)**. However, the results on **(1)** are useful in order to address the two major concerns of CAP when the input is relatively small: (1a) Will the overhead of the preprocessing (component abstraction + macro generation) decrease the coverage compared to the base planner? (1b) Will solution quality suffer significantly?

The IPC2011 results in Table 1 show that the coverage of CAP(FF) increased over FF (172 vs 178). Similarly,

Domain	FF	CAP(FF)	CAP (FF, pre15)	$\frac{c(\text{CAP(FF)})}{c(\text{FF})}$ mean±sd	FD/lama	CAP(FD/lama)	CAP (FD/lama, pre15)	$\frac{c(\text{CAP(FD/lama)})}{c(\text{FD/lama})}$ mean±sd	Probe	CAP(Probe)	CAP (Probe, pre15)	$\frac{c(\text{CAP(Probe)})}{c(\text{Probe})}$ mean±sd	FD/h^{cea}	CAP(FD/h^{cea})	CAP (FD/h^{cea}, pre15)	$\frac{c(\text{CAP(FD/}h^{\text{cea}}\text{)})}{c(\text{FD/}h^{\text{cea}})}$ mean±sd
barman-ipc11(20)	0	20	20	-	20	20	20	.85±.06	20	20	20	1.00±.15	0	10	10	-
elevators-ipc11(20)	20	20	20	2.03±.52	20	20	20	1.42±.40	17	19	20	1.44±.30	3	3	3	.91±.08
floortile-ipc11(20)	10	4	4	.99±.03	4	5	5	.99±.06	4	3	3	.87±.00	6	5	5	.95±.08
nomystery-ipc11(20)	8	6	6	.98±.11	13	7	7	1.08±.03	9	6	6	.99±.04	6	6	6	1.04±.05
openstacks-ipc11(20)	20	0	0	-	20	5	15	.67±.07	12	10	11	1.19±.19	0	0	0	-
parcprinter-ipc11(20)	20	18	18	.99±.03	14	20	20	.96±.07	13	18	18	.91±.09	14	13	13	1.00±.00
parking-ipc11(20)	5	6	7	1.05±.11	20	16	14	1.13±.24	13	10	9	1.01±.14	9	9	9	1.00±.00
pegsol-ipc11(20)	20	20	20	1.00±.00	20	20	20	1.00±.02	20	20	20	1.00±.02	20	20	20	1.00±.00
scanalyzer-ipc11(20)	19	19	19	1.00±.00	18	18	18	1.03±.12	17	17	17	1.00±.01	17	17	17	1.00±.00
sokoban-ipc11(20)	15	15	15	1.02±.08	19	19	19	.98±.09	16	15	15	1.01±.07	13	13	13	1.05±.19
tidybot-ipc11(20)	13	13	13	1.00±.00	15	15	15	1.00±.00	18	18	18	1.00±.09	18	18	17	1.00±.00
transport-ipc11(20)	8	14	14	1.38±.26	15	19	18	1.58±.24	13	15	14	1.24±.14	6	6	6	1.23±.15
visitall-ipc11(20)	4	4	4	1.00±.00	19	17	19	1.00±.00	18	13	15	1.00±.04	3	3	3	1.00±.00
woodworking-ipc11(20)	10	19	19	1.00±.03	20	20	20	1.05±.05	19	20	20	1.02±.05	18	20	20	1.04±.11
Sum	172	178	179	1.17±.42	237	221	230	1.07±.26	209	204	206	1.06±.19	133	143	142	1.01±.09
assembly-mixed-large(20)	4	20	20	.71±.08	4	19	19	.87±.15	4	20	20	.66±.14	2	2	2	.63±.08
barman-large(20)	0	9	8	-	3	19	20	.98±.02	4	7	10	1.09±.03	0	0	0	-
elevators-large(20)	3	7	7	2.54±.52	11	20	18	1.65±.24	0	11	12	-	0	0	0	-
floortile-large(20)	5	3	3	1.23±.19	2	2	2	1.04±.06	2	2	2	1.08±.08	2	3	3	.92±.01
nomystery-large(20)	2	1	1	.87±.00	5	3	3	1.00±.03	0	0	0	-	2	2	2	1.03±.00
openstacks-large(21)	19	0	0	-	21	0	18	-	0	0	0	-	0	0	0	-
parking-large(20)	2	1	1	1.00±.00	19	9	9	1.00±.00	0	0	0	-	2	3	3	1.00±.00
tidybot-large(20)	10	10	10	1.00±.00	9	9	9	1.00±.00	13	12	12	1.00±.00	7	6	7	1.00±.00
transport-large(20)	0	2	2	-	3	14	13	1.62±.14	5	6	7	1.26±.17	0	1	1	-
visitall-large(20)	0	0	0	-	20	0	13	-	0	0	0	-	0	0	0	-
woodworking-large(20)	3	0	0	-	0	11	11	-	4	4	3	1.01±.02	0	13	13	-
Sum	48	53	52	1.18±.60	97	106	135	1.20±.35	32	62	66	1.01±.19	15	30	31	.94±.13
barman-ipc11-learning-test(30)	0	30	30	-	4	29	28	.90±.03	6	28	28	.91±.04	0	0	0	-
blocksworld-ipc11-learning-test(30)	6	6	6	1.00±.00	26	27	26	1.00±.00	19	17	20	1.02±.11	1	1	1	1.00±.00
depots-ipc11-learning-test(30)	4	3	4	.97±.04	0	0	0	-	29	30	29	1.00±.09	0	0	0	-
gripper-ipc11-learning-test(30)	0	0	0	-	0	0	0	-	0	0	0	-	0	0	0	-
parking-ipc11-learning-test(30)	1	1	1	1.00±.00	14	9	10	1.00±.00	1	5	4	1.00±.00	0	0	0	-
rover-ipc11-learning-test(30)	2	5	5	.99±.01	29	17	22	1.03±.08	18	16	18	1.06±.11	0	0	0	-
satellite-ipc11-learning-test(30)	2	4	4	-	5	2	2	1.00±.00	0	0	0	-	0	0	0	-
spanner-ipc11-learning-test(30)	0	0	0	-	0	0	0	-	0	0	0	-	0	0	0	-
tpp-ipc11-learning-test(30)	0	20	20	-	16	29	29	1.45±.12	9	10	10	1.24±.08	1	0	0	-
Sum	15	69	70	.99±.02	94	113	117	1.10±.20	82	106	109	1.04±.13	2	1	1	1.00±.00

Table 1: Coverage results (#) and cost comparison on IPC2011 seq-sat instances, its large instances and IPC2011 learning track test instances. Participants are 4 baseline planners, their CAP variants and the CAP variants with 15[min] preprocessing time limit (pre15). Cost ratios include only those instances that were solved by baseline and CAP configurations.

Domain	FF	CAP(FF)	CAP (FF, pre7.5)	Chrpa baseline FF	Wizard FF	MUM FF	FD/lama	CAP(FD/lama)	CAP (FD/lama, pre7.5)	Chrpa baseline FD/lama	Wizard FD/lama	MUM FD/lama	Probe	CAP(Probe)	CAP (Probe, pre7.5)	Chrpa baseline Probe	Wizard Probe	MUM Probe
barman-ipc11-learning-test(31)	0	30	29	0	0	0	3	28	29	0	0	0	2	20	30	1	1	1
blocksworld-ipc11-learning-test(60)	6	5	6	0	0	0	26	24	26	18	0	27	19	19	19	20	18	30
depots-ipc11-learning-test(60)	4	3	3	1	2	4	0	0	0	3	3	3	30	30	30	30	30	30
gripper-ipc11-learning-test(56)	0	0	0	0	0	25	0	0	0	0	0	26	0	0	0	0	0	0
parking-ipc11-learning-test(37)	1	1	1	7	0	7	11	7	9	4	4	4	3	0	0	3	3	3
rover-ipc11-learning-test(59)	1	2	3	0	0	0	24	4	10	6	6	29	16	12	16	20	20	11
satellite-ipc11-learning-test(48)	1	4	4	0	0	7	1	0	0	2	2	18	0	0	0	0	0	0
spanner-ipc11-learning-test(30)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
tpp-ipc11-learning-test(46)	0	20	20	0	0	0	12	29	27	13	13	16	6	10	9	10	6	15
Sum	13	65	66	8	2	43	77	92	101	46	28	123	76	91	104	84	78	90

Table 2: Coverages of our baselines, CAP, CAP+7.5 preprocessing limit (pre7.5), Chrpa's baselines, Wizard and MUM (Chrpa, Vallati, and McCluskey 2014). All planners are run on IPC2011 learning track test instances, for 15[min] with 4[GB] memory. MUM and Wizard have additional 15[min] for learning.

CAP(FD/h^{cea}) have significantly increased coverage compared to FD/h^{cea}. On the other hand, CAP(FD/lama) and CAP(Probe) has both worse coverage compared to FD/lama and Probe.

With respect to plan cost, there tends to be a slight overall penalty. For example, $c(\text{CAP(FF)})/c(\text{FF})$ shows the ratio of plan costs of CAP(FF) to FF, on problems solved by both, and it is 1.17 on average with a standard deviation of 0.42. However, as would be expected, results vary per domain, and we observed that the cost has improved in some domains: namely, barman, floortile, openstacks (on FD/lama), parcprinter.

Our main results are the lower half of Table 1, which shows the results for **(2) large instances** and **(3) IPC2011 learning track testing instances**. In **(2)**, we evaluated CAP on assembly-mixed, the original motivating domain for this work, as well as larger instances of IPC2011 domains obtained by the same generator as in the competition, but with significantly larger parameters (see Table 3 for a comparison). We could not locate generators for pegsol, parcprinter, scanalyzer, sokoban, so their results are not shown here. However, extrapolating from the IPC2011 instances of these domains, it seems safe to say that the overall trends would not be significantly affected even if large instances of these domains were available. In **(3)**, we used the test instances of IPC2011 learning track. We repeat that this experiment is run with 30[min] time limit and 2[GB] memory limit, unlike the original learning track setting where 15[min] learning + 15[min] planning + 4[GB] memory is allowed.

Overall, CAP variants obtained significantly higher coverages. We see major improvements in barman, elevators, transport, woodworking, tpp, while there is a severe degradation in openstacks, parking and visitall.

domain	IPC2011 instances	large instances
assembly-mixed	n/a (not an IPC2011 domain)	1-20 product A, 1-20 product B
barman	10-15 shots, 8-11 cocktails	47-78 shots, 47-71 cocktails
elevators	16-40 floors 13-60 passengers	40 floors 62-137 passengers
floortile	3x5 - 8x8 tiles, 2-3 robots	Same range, but different distribution
nomystery	6-15 locations & packages	8-24 locations & packages
openstacks	50-250 stacks	170-290 stacks
parking	10-15 curbs 20-30 cars	15-18 curbs 28-34 cars
tidybot	81-144 area, 5-15 goals	144-169 area, 8-20 goals
transport	50-66 nodes,20-22 packages	66 nodes, 20-60 packages
visitall	12-50 grids	61-80 grids
woodworking	3-13 parts,1.2-1.4 woods	34-81 parts,1.2-1.4 woods

Table 3: Large instances: some characteristics compared to IPC2011 instances.

Improving CAP by Limiting the Preprocessing Time CAP would perform well on domains that can be decomposed into serializable subproblems. Thus, on domains without this decomposable structure, CAP will not yield any benefits – the time spent by the CAP preprocessor analyzing the domain and searching for component macros will be wasted. According to Table 4, which shows the fraction of time spent on preprocessing by CAP on the large instances, CAP sometimes spends the majority (85-100%) of its time on preprocessing. While the preprocessing time varies, there are essentially three cases: 1) There are few, if any, component problems so preprocessing is fast (e.g. tidybot); 2) Although useful macros are discovered, the preprocessing consumes too much time, leaving insufficient time for the MainPlanner to succeed (e.g. elevators); or 3) large preprocessing time is wasted because the macros are not useful to the MainPlanner (e.g. openstacks).

A simple approach to alleviate failures due to cases 2 and 3 above is to limit the time CAP spends on its macro generation phase. As shown in Table 1, running CAP with a 15 minute preprocessing limit (50% of the 30-minute overall limit) dramatically improves coverages for most of the CAP variants (except CAP(FD/h^{cea}) on IPC2011 seq-sat and CAP(FF) on Large). The 15-minute cutoff is arbitrary and has not been tuned. This shows that a reasonable preprocessing limit allows CAP to discover some useful macros, while leaving sufficient time for the MainPlanner to exploit these macros and solve the problem more effectively than the MainPlanner by itself (addressing case 2). Furthermore, on problems where CAP can not identify useful macros, this prevents CAP from wasting all of its time on a fruitless search for macros (addressing case 3).

Domain	CAP(FF) mean±sd	CAP(FD/lama) mean±sd	CAP(Probe) mean±sd	CAP(FD/h^{cea}) mean±sd
assembly-mixed-large(20)	.84±.10	.83±.05	.60±.20	.07±.19
barman-large(20)	.60±.42	.90±.01	.87±.14	.91±.12
elevators-large(20)	.84±.16	.60±.13	.42±.35	.14±.03
floortile-large(20)	.16±.27	.05±.09	.06±.17	.07±.14
nomystery-large(20)	.08±.12	.34±.32	.28±.39	.27±.24
openstacks-large(21)	.94±.10	1.00±.00	1.00±.00	1.00±.00
parking-large(20)	.56±.49	.70±.33	.95±.02	.57±.48
tidybot-large(20)	.06±.11	.01±.00	.07±.21	.01±.02
transport-large(20)	.56±.44	.51±.21	.46±.36	.23±.26
visitall-large(20)	1.00±.00	1.00±.00	1.00±.00	1.00±.00
woodworking-large(20)	.26±.12	.67±.20	.31±.09	.76±.17
Mean	.52±.42	.57±.37	.53±.41	.43±.42

Table 4: The fraction of time spent on preprocessing (out of 30 min. total runtime) on large problem instances.

Comparison against Previous Macro Based Planners/Learners

To show how CAP performance compares to the existing macro-based methods, we first compared CAP against Marvin. Table 5 shows the coverages of FF, CAP(FF) and Marvin on **(1,2,3)**, with 30[min] and 2[GB] resource constraint. We chose CAP(FF) for comparison to Marvin because Marvin is based on FF. Neither FF or Marvin are able to handle action-costs, so we fed the unit-cost version of the domain and the problem. We also removed some domains because the latest 32bit binary of Marvin runs into segmentation faults or emits invalid plans. Interestingly, CAP(Marvin) and CAP(Marvin, pre15) improves upon Marvin on some domains, indicating that CAP is complementary to Marvin's plateau escaping macro.

Next, in Table 2, we compared CAP with the results of (Chrpa, Vallati, and McCluskey 2014). Experiments are run on the testing instances of IPC2011 learning track, with

Domain	FF	CAP(FF)	CAP (FF, pre15)	Marvin	CAP (Marvin)	CAP (Marvin, pre15)
barman-ipc11(20)	0	20	20	16	20	20
floortile-ipc11(20)	10	4	4	2	1	1
nomystery-ipc11(20)	8	6	6	8	3	4
openstacks-ipc11(20)	20	0	0	19	9	12
pegsol-ipc11(20)	20	20	20	20	20	20
scanalyzer-ipc11(20)	19	19	19	19	20	20
sokoban-ipc11(20)	15	15	15	10	9	10
tidybot-ipc11(20)	13	13	13	14	14	14
transport-ipc11(20)	8	14	14	0	13	13
visitall-ipc11(20)	4	4	4	0	1	1
Sum	117	115	115	108	110	115
barman-large(20)	0	9	8	0	20	19
floortile-large(20)	5	3	3	3	2	2
nomystery-large(20)	2	1	1	3	0	0
openstacks-large(21)	19	0	0	11	0	5
tidybot-large(20)	10	10	10	9	10	10
transport-large(20)	0	2	2	0	2	3
visitall-large(20)	0	0	0	0	0	0
Sum	36	25	24	26	34	39
barman-ipc11-learning-test(30)	0	30	30	1	30	30
blocksworld-ipc11-learning-test(30)	6	6	6	12	10	11
depots-ipc11-learning-test(30)	4	3	4	0	1	1
satellite-ipc11-learning-test(30)	2	4	4	0	11	10
tpp-ipc11-learning-test(30)	0	20	20	1	27	27
Sum	12	63	64	14	79	79

Table 5: Coverages of FF, CAP(FF), CAP(FF,pre15), Marvin, CAP(Marvin) and CAP(Marvin,pre15), on 30[min], 2[GB] experiments. In this table, elevators, woodwoking, assembly-mixed, parcprinter, parking, gripper, rover, spanner are not counted for all 6 configurations because Marvin, CAP(Marvin) and CAP(Marvin, pre15) obtained 0 coverages possibly due to the aforementioned bug. FF family has solved them successfully (same results as in Table 1.)

4[GB] memory limit. MUM and Wizard are given 15[min] for learning *plus* 15[min] for solving. CAP variants are given only the solving phase of 15[min]. Also, CAP variants with 7.5[min] preprocessing time limit within this 15[min] is shown as pre7.5. Thus, this experiment is completely independent from Table 1. CAP, including its preprocessing (macro generation), should run entirely in the solving phase. First of all, CAP is not able to learn from the different training instances. Also, using the test instances for learning is a *post hoc theorization* and does not make sense.

Due to the differences in CPU, RAM, execution environments or parameters etc., coverage results in (Chrpa, Vallati, and McCluskey 2014) should not be directly compared to ours. Their experiments are run on a "3.0 GHz machine" (2014, p.5) while we use 2.33GHz Xeon E5410. Therefore, the baseline (control) coverages differ between ours and theirs, e.g., our baseline FD/lama coverage sum is 75 (all plans verified using the Strathclyde VAL plan validation tool), while "Chrpa baseline FD/lama" solved 46 instances.

In Figure 2, CAP improves performance in different domains than where MUM exhibits its performance – In particular, CAP performs well in barman and TPP, while MUM performs well in blocksworld, gripper, rover and satellite. Based on these results, it is safe to say CAP and MUM is complementary. The combination of CAP with MUM is an avenue of future work.

How CAP macro works

So far, we claimed the CAP's peculiarity based solely on the difference of domains where the coverage was improved. Now we show where it comes from.

First, we measured the length of the macros found (i.e. including those not used) and the macros used during the search, shown in Figure 3. This clearly shows that CAP finds both long and short macros: There are > 1000 short macros of length 2 at the same time many macros with length > 100. As expected, the size and the shape of the curve depends on the domain. If we limit the macros to those used in the result plan, the points decrease significantly (i.e. they were not used at all). However, a large number of long and short macros still remain. We saw the similar distribution for Large instances and IPC2011 Learning instances, but they are omitted due to space.

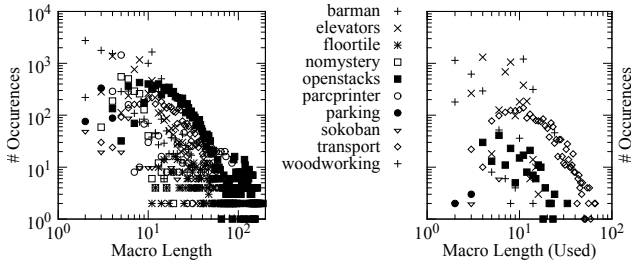

Figure 3: (Left) Distribution of the length of macros found by CAP(FD/lama) in IPC2011 seq-sat instances. x-axis shows the macro length and y-axis shows how many macros are found. Both axes are in logarithmic scale. (Right) Distribution of the length of macros *used in the result plan* of the same instances.

We also claimed that, although large amount of long macros are found and used, it does not harm the planner performance by increasing the branching factor because they are always zero-ary actions, and the situation where the macro is applicable is very limited (but once applied, it is very effective). We support these claims in Figure 4 (Left). We plotted the node expansion of baseline planner X in the x-axis and that of the main search of CAP(X) in the y-axis, where $X \in \{\text{FF}, \text{FD/lama}, \text{FD/}h_{\text{cea}}, \text{Probe}\}$ (all included) and each point represents an IPC2011 seq-sat instance solved by both X and CAP(X). Now, while some domains show the increase, in many domains, the number of expansion turns out to be neutral or even *dramatically reduced*. This results suggests that these rarely applicable zero-ary macros do benefits planners' performance, despite the common conception of macro utility problems.

Also, Figure 4 (Right), which plots the number of ground actions on the same set of instances solved by the baseline planner $X \in \{\text{FF}, \text{Probe}\}$ on x-axis, and that of the main search of CAP(X) on y-axis, supports that the number of actions increased by CAP are negligeble and does not hurt the initialization phase of the planners.

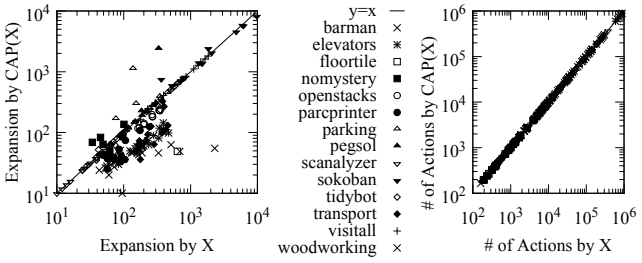

Figure 4: Each point represents IPC2011 seq-sat instances solved by baseline planner X and its CAP variant CAP(X). x-axis represents the baseline planner X and y-axis represents the main search of CAP(X). Coordinate values indicate: (Left) the number of node expansion, where $X \in \{\text{FF, FD/lama, FD/}h_{\text{cea}}\text{, Probe}\}$. (Right) the number of instantiated ground actions where $X \in \{\text{FF, Probe}\}$.

Related Work

Asai and Fukunaga (2014) developed ACP, a system which automatically formulates an efficient cyclic problem structure from a PDDL problem for assembling a single instance of a product. ACP focuses on identifying efficient cyclic solutions for mass manufacturing multiple instances of one particular product. In CAP terminology, ACP assumes that its input contains exactly 1 kind of "component" and nothing else. Thus, ACP can not handle *heterogeneous*, repetitive problems, e.g., a manufacturing order to assemble 20 instances each of Widget A and Widget B. CAP is therefore a more general approach.

Component abstraction in CAP is based on the algorithm originally introduced by (Botea, Müller, and Schaeffer 2004) for MacroFF/CA-ED. CAP differs fundamentally from CA-ED in how to use the component abstraction for macro generation. CA-ED only uses it in a brute-force enumeration of short action sequences (macros) relevant to the components, therefore most such macros are useless and need to be filtered. Their macros are lifted because they aims at finding reusable macros across problems (DCK), and could not get longer without affecting the branching factor. In contrast, since CAP aims at satisficing setting, CAP macros are designed to be problem-specific. The search for macros can therefore be **goal-driven** — we extract/solve/encode subgoals/subproblems/subplans relevant to the components, into macros. They contain useful information of the problem, eliminating the needs for filtering. Also, CAP macros are zero-ary (i.e. not lifted) and does not increase the branching factor, with allows for the longer macros.

CAP can be viewed as a method for identifying and connecting a tractable set of waypoints in the search space. This is somewhat related to Probe (Lipovetzky and Geffner 2011), a best first search augmented by "probes" that try to reach the goal from the current state by quickly traversing a sequence of next unachieved landmarks in a "consistent"(Lipovetzky and Geffner 2009) greedy chain. CAP and Probe work at different levels of abstraction: While the landmarks are single propositions (Probe does not handle disjunctive landmarks), CAP identifies serializable subproblems (Component Problems), each of which contains *a set of subgoals* and requires a significant amount of search by the ComponentPlanner. As shown in Table 1, the class of problems for which CAP is designed cannot be easily solved by Probe, but more interestingly, CAP is complementary to Probe. CAP(Probe), which uses Probe as both the ComponentPlanner and MainPlanner, performs significantly better than Probe by itself.

CAP is an offline approach which statically analyzes a problem to generate an enhanced domain. In contrast, online macro learning approaches such as Marvin (Coles and Smith 2007) embed macro learning into the search algorithm. An advantage of offline approaches is modularity – they can easily collaborate with various planners. On the other hand, online approaches may allow more effective, dynamic reasoning about macros. Embedding component abstraction as an online macro learning mechanism is a direction for future work.

CAP can be viewed as an extraction method of hierarchical structure, restricted to 2 layers. In contrast, decomposition in HTN planning(Erol, Hendler, and Nau 1994) is done by human experts. Also, previous work on extracting HTN (Nejati, Langley, and Konik 2006; Hogg, Munoz-Avila, and Kuter 2008) aims at learning from expert plans. While CAP requires the type information to be encoded by the domain creators, this is not restrictive because TIM(Fox and Long 1998) can infer types automatically. Also, typeless domains are not unrealistic in practice, because they may be converted from some other computationally intractable problems and may lack the type information.

Conclusions

We proposed CAP, an approach to solving large, decomposable problems. CAP uses static analysis of a PDDL domain to decompose the problem into components. Plans for solving these components are solved independently and are converted to large macros which are added to the domain.

We showed that CAP can be used to find satisficing solutions to large problems (with a particular type of decomposable structure) that are beyond the reach of current, state-of-the-art domain-independent planners. CAP is implemented as a wrapper for those planners, and was successfully applied to 4 different combinations of planners and heuristics. We experimentally compared CAP with previous macro based approaches and showed that they are orthogonal to CAP. Unlike those approaches, CAP finds useful macros without filtering nor increase of branching factor.

So far, we have focused on achieving high coverage. Future work will investigate plan inefficiency in-depth, including the use of fast plan optimization techniques in a postprocessor to improve plan efficienty (c.f. (Nakhost and Müller 2010; Chrpa, McCluskey, and Osborne 2012)).

Finally, one interesting aspect of CAP is the possibility of combining different plannners as the ComponentPlanner and MainPlanner (Figure 1). Preliminary experiments show that using a fast, greedy planner (e.g., FF) as the ComponentPlanner and a more deliberative planner (e.g., FD/lama) as MainPlanner can improve performance on some domains. This will be explored further in future work.

References

Asai, M., and Fukunaga, A. 2014. Fully Automated Cyclic Planning for Large-Scale Manufacturing Domains. In *Proceedings of the International Conference of Automated Planning and Scheduling(ICAPS)*.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *J. Artif. Intell. Res.(JAIR)* 24:581–621.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Using Component Abstraction for Automatic Generation of Macro-Actions. In *Proceedings of the International Conference of Automated Planning and Scheduling(ICAPS)*, 181–190.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012. Optimizing Plans through Analysis of Action Dependencies and Independencies. In *Proceedings of the International Conference of Automated Planning and Scheduling(ICAPS)*.

Chrpa, L.; Vallati, M.; and McCluskey, T. L. 2014. MUM: A Technique for Maximising the Utility of Macro-operators by Constrained Generation and Use. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 65–73.

Coles, A., and Smith, A. 2004. Marvin: Macro Actions from Reduced Versions of the Instance. In *Proceedings of the 4th International Planning Competition (IPC)*.

Coles, A., and Smith, A. 2007. Marvin: A Heuristic Search Planner with Online Macro-Action Learning. *J. Artif. Intell. Res. (JAIR)* 28:119–156.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN Planning: Complexity and Expressivity. In *AAAI*, volume 94, 1123–1128.

Fox, M., and Long, D. 1998. The Automatic Inference of State Invariants in TIM. *J. Artif. Intell. Res.(JAIR)*.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.(JAIR)* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation through Heuristic Search. *J. Artif. Intell. Res. (JAIR)* 14:253–302.

Hogg, C.; Munoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *AAAI*, 950–956.

Korf, R. E. 1987. Planning as Search: A Quantitative Approach. *Artificial Intelligence* 33(1):65–88.

Lipovetzky, N., and Geffner, H. 2009. Inference and Decomposition in Planning using Causal Consistent Chains. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*.

Lipovetzky, N., and Geffner, H. 2011. Searching for Plans with Carefully Designed Probes. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*.

Nakhost, H., and Müller, M. 2010. Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, 121–128.

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning Hierarchical Task Networks by Observation. In *Proceedings of the 23rd international conference on Machine learning*, 665–672. ACM.

Newton, M. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning Macro-Actions for Arbitrary Planners and Domains. In *Proceedings of the International Conference of Automated Planning and Scheduling(ICAPS)*, 256–263.