Solving Large-Scale Planning Problems by Decomposition and Macro Generation

Masataro Asai and **Alex Fukunaga**
Graduate School of Arts and Sciences
The University of Tokyo

Abstract

Large-scale classical planning problems such as factory assembly problems pose a significant challenge for domain-independent planners. We propose a macro-based planner which automatically identifies subproblems, generate macros from subplans and integrate the subplans by solving the whole augmented problem. 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.

One promising approach to solve this kind of problem is a broad class of planners called Factored Planning (FP) (Amir and Engelhardt 2003; Brafman and Domshlak 2006), which provides a framework that tries to decompose a problem into easy subproblems, solve them independently, and combine those results. FP is potentially a very powerful framework which can subsume HTN planning (Erol, Hendler, and Nau 1994), where the original problem is decomposed by human experts. While the previous work on FP uses Causal Graph for decomposition, they have been limited to problems with specific structures in their causal graphs, and FP has not yet been shown to be practical in general.

In this paper, we propose **C**omponent **A**bstraction **P**lanner (CAP), an approach to solving large scale planning problems using macro operators, which can be viewed as a type of FP. For example, in the assembly domain mentioned above, each subproblem corresponds to the assembly of a single product such as Widget A or Widget B. The subproblems are solved using standard, domain-independent planners, solutions to these subproblems are converted to macros operators, and the whole problem augmented by the macros is solved by a standard planner.

The key difference between CAP and all previous macro-based methods to our knowledge is that CAP does not seek *reusable* (highly applicable) macros. 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 learning scenarios such as the 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. In such a learning setting, macros should encapsulate knowledge that is *reusable* across the problems in a domain. Marvin (Coles and Smith 2007) learns macros in a standard (IPC2011 satisficing track) setting, but it also seeks macros that are reusable within the same problem instance (Coles and Smith 2007, page 10). CAP differs from all of these previous approaches. First, CAP targets a standard

(IPC2011 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 2014, p.8) because such recipes are not reusable across instances. However, a Barman instance can be decomposed into easy, independent cocktail mixing problems, and our results show that CAP significantly increases coverage on Barman. Also, when all recipes are different, searching for reusable macros *within the same problem* will fail. If a bartender is given orders to prepare different cocktails A, B and C, the solution to the subproblem of mixing cocktail A is not reusable in mixing B and C. Thus, reusability (high applicability) is unnecessary for CAP macros to be useful. Our experimental results show that the decomposition itself is highly effective.

The second major difference between CAP and previous approaches is its ability to find a large number of informative, very long macros, without the need for macro-filtering or upper bounds on macro length. All previous systems such as MacroFF, Wizard and MUM filter their macros and/or apply macro length bounds in order to prevent the search algorithm branching factor from exploding. For example, MacroFF / CA-ED generates 2-step macros, and MUM finds macros "of length of 2 or 3, occasionally (...) 5". Jonsson (2007) finds long macros but is limited to domains where all operators are unary and the Causal Graph is acyclic. CAP is able to avoid these limitations by using completely grounded, *nullary* macros that encapsulate solutions to specific subproblems.

Finally, although some macro-based systems such as Marvin, which generates macros to escape plateaus in the search space, are closely coupled with the search algorithm used by the planner, CAP is implemented as a wrapper for any PDDL (STRIPS + action cost) planner and as we show below, it can be easily combined with various planners, including FF (Hoffmann and Nebel 2001), configurations of Fast Downward (Helmert 2006), and Probe (Lipovetzky and Geffner 2011). CAP can even be used as a wrapper for previous macro-based approaches such as Marvin.

The resulting system 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 the state-of-the-art Fast Downward planner (LAMA2011 configuration) 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. The CAP source repository is at **`https://github.com/guicho271828/CAP`** .

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

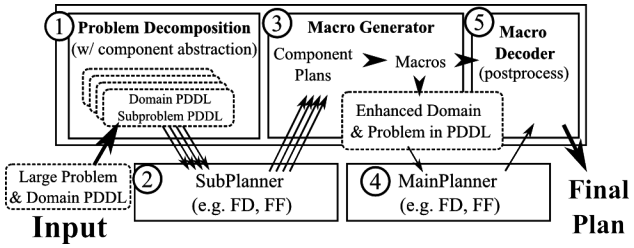

Figure 1: CAP System Overview. SubPlanner and MainPlanner are domain-independent planners, e.g., FD/lama (Helmert 2006), FF (Hoffmann and Nebel 2001). They can be the same planner, or different planners.

Overview of CAP

Given a large-scale problem such as the previously described heterogeneous factory assembly and Barman instances, we propose an approach that automatically decomposes those problems into independent subproblems and sequences their solutions. Figure 1 gives an overview of our overall approach, **C**omponent **A**bstraction **P**lanner (CAP). CAP performs the following steps:

1. *Problem Decomposition*: Perform a static analysis of the PDDL problem in order to identify the independent subproblems. We run a Component Abstraction algorithm to identify the *components* in a problem, build a component task (subproblem) for each component.
2. *Generate Subplans with SubPlanner*: Solve the subproblems with a domain-independent planner (SubPlanner).
3. *Macro generation*: For each subplan, concatenate all of its actions into a single, ground (nullary) macro operator.
4. *Main Search by MainPlanner*: Solve the augmented PDDL domain (including macros) with a standard domain-independent planner (MainPlanner).
5. *Decoding*: Finally, any macros in the plan found by MainPlanner are decoded into the primitive actions.

Problem Decomposition The decomposition of a problem into a set of easier subproblems is done by identifying *component tasks*. Component tasks are generated from *abstract components*, which are extracted using the Component Abstraction algorithm originally used in MacroFF/CA-ED (Botea, Müller, and Schaeffer 2004). CA-ED only uses components in a brute-force enumeration of short (2-step) macros, so most macros found by CA-ED are useless and need to be filtered. In contrast, CAP takes the idea of components much further: A component task consists of the initial and goal propositions relevant to 1 component, and solving a component task results in a single macro that solves an entire subproblem.

Component Abstraction (Botea et al. 2004) Component Abstraction (CA) first builds a *static graph* of the problem and clusters it into disjoint subgraphs (components). This algorithm requires a typed PDDL, but this is not restrictive because the types can be automatically added to untyped domains using methods such as TIM (Fox and Long 1998).

The static graph of a problem 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.

Figure 2 illustrates the static graph of a simple assembly problem where the task is to assemble 2 widgets by assembling a0 and a1 with parts b0 and b1, respectively. The fill pattern of each node indicates its type, e.g. b0 and b1 of type part are wave-patterned (▒). The edge (a0, b0) corresponds to a predicate (part-of a0 b0) specifying that "b0 is a part-of a0", which statically declares that b0 should be assembled with a0, as opposed to, e.g., "a0 and b0 are assembled", which may be modified by an action "assemble". Also, the edge (b0, red) indicates a static fact that "part b0 can be painted in red" (b0 is allowed to be painted red somewhere in the plan), as opposed to (painted b0 red), which may be modified by some actions like "paint" or "clean". The goal of this problem is (assembled a0 b0) and (assembled a1 b1).

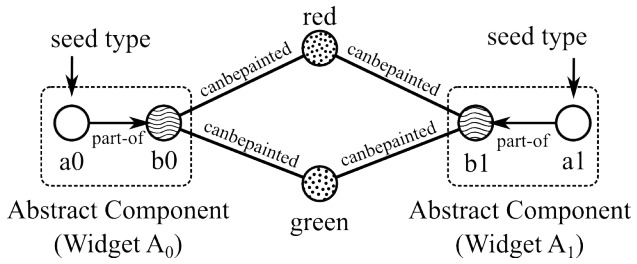

Figure 2: Example of a static graph and components.

The planner usually does not know that there are two assembly processes that can be performed independently. We would like CA to identify the widgets in this problem, shown as A_0, A_1, from the static graph. The CA algorithm, slightly modified from (Botea et al. 2005), works as follows. 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 (below, the text in brackets "[]" describes the corresponding behavior in Figure 2):

1. If T is empty, return R. Otherwise, pop one *seed type* s from T [The type of a0: ○ 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 ($b \notin c', \forall c' \in C, c' \neq c$). An n-ary predicate $p(x_1, \ldots x_n)$ is treated as a combination 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 $R = R \cup C$, 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 $\{A_0 =$ (a0 b0), $A_1 =$ (a1 b1)$\}$]. However, if the resulting components share any part of the structure, we discard newly merged nodes [extending these components further by merging ▒s results in (a0 b0 red green),(a1 b1 red green) which share red and green, so this extension is discarded]. Finally, go back to step 3.

The resulting components form an equivalence class called an *abstract type*. Components are of the same abstract type when their clustered subgraphs are isomorphic. This allows the planner to detect something like "Widget-A Type", a structure shared among A_0 and A_1, analogous to the notion of *Class* in object-oriented programming.

Our modified CA algorithm tries all seed types s as long as some object of type s is referenced by the goal conditions, while Botea et al. (2004) selects s randomly regardless of the goal. In addition, while their procedure stops and returns C immediately when some fully extended C meets some (heuristically chosen) criteria, we collect C into R and use all results of the iterations. This allows us to collect all components necessary to achieving the goal.

Component Tasks While (Botea, Müller, and Schaeffer 2004) generated short macros by using abstract components to identify "related" actions, our novel approach fully exploits the structure extracted by Component Abstraction. For an abstract component X, we can define a *component task* as a triple consisting 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 (non-static) version of fringe facts. A *fluent fringe fact* $f(a, b)$ of X is defined as a fluent (a predicate that can be added or removed by some action) such that 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 fluent fringe facts in the initial state and the goal condition as *init* (X) and *goal* (X), respectively. For example, the initial state of the problem in Figure 2 may contain (painted b0 white), a fluent which takes b0 $\in A_0$ and may be removed by some actions like paint. Similarly, (assembled a0 b0) in the goal is a fluent fringe fact of A_0, and therefore a goal condition specific to A_0.

Next, since a component task is a compact representation of a subproblem, we can expand it into a PDDL problem. 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 subproblem $\Pi_X = \langle \mathcal{D}, O_X, I_X, G_X \rangle$ is defined as follows:

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

Informally, O_X contains X, but does not contain the siblings of X, the components of the same abstract type. O_X also contains E_X, which can be seen as environment-like objects that are not part of any such siblings e.g. painting machines used by all widgets that needs painting. In I_X, ill-defined propositions (referencing the removed objects) are removed.

We solve Π_X with SubPlanner (Figure 1), which is a domain-independent planner such as Fast Downward (Helmert 2006) or FF (Hoffmann and Nebel 2001), using a short time limit (30 seconds for all experiments in this paper). Π_X 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 unsatisfiable or ill-defined (e.g., G_X contains references to removed objects). If SubPlanner 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 generated the subplans which are solutions to corresponding subproblems. In general, it is not possible to solve the whole problem by simply concatenating these subplans, because each subplan is based on local reasoning about some projection of the original search space. Previous work in factored planning seeks a tree-decomposition of the problem description and uses a bottom-up planning algorithm to combine the subplans into a complete plan (Amir and Engelhardt 2003). In contrast, CAP takes a "best-effort" approach which seeks plan fragments that *might* be composable.

Instead of seeking plan fragments that are guaranteed to be composable into a complete plan, we generate macro operators based on the subplans, and add them to the original PDDL domain. This augmented domain is then solved using a standard domain-independent planner (MainPlanner in Figure 1). MainPlanner uses those macros for solving each subproblem in 1 step, inserting additional actions as necessary. If the resulting plan uses macro operators, they are later mapped back to the corresponding ground instances of the primitive actions in the original domain. On the other hand, it is possible that some (or all) macros are not used at all, and the planner relies heavily on the primitive actions in the original domain.

A subplan is converted to a 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 subplan results in a single, ground macro operator that represents the application of the plan. Feasibility (internal consistency) of CAP macros is clearly guaranteed as long as the SubPlanner is a sound planner.

Since each macro generated by CAP corresponds to the solution to a single component problem, all merged actions are fully grounded. Our macros are always *nullary*, unlike traditional macro operators which are the combinations of several parametrized (lifted) actions. This has several important implications: First, they do not significantly increase the number of ground actions. Current forward-search based planners tend to instantiate ground actions from action schema, but their number increases exponentially with the number of parameters. Lifted macros suffer from this explosion because merging actions into macros creates actions with more parameters (in our preliminary experiments, long lifted macros caused the input parser/translator to exhaust memory), while nullary macros produce only 1 ground instance each. Second, since they are grounded and are applicable to the limited situations, they do not increase the effective branching factor significantly. Finally, the overwhelming majority of ground instances of lifted macros are useless. Since a subplan is inherently specialized to the subproblem, each subplan is less likely to be useful in the other contexts. In contrast, our nullary macros do not produce ground actions with futile combinations of parameters.

Experimental Results

All experiments below were 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 SubPlanner (for component planning) and the MainPlanner (for solving the macro-augmented problem generated by CAP). It is possible for SubPlanner and MainPlanner to be different planners (Figure 1): Some planners may be better suited as the SubPlanner rather than the MainPlanner, and vice versa, so mixed configurations, e.g., CAP(FF +FD/lama), which uses FF as SubPlanner and FD/lama as MainPlanner, are possible. However, in this paper, for simplicity, we focus on configurations where SubPlanner and MainPlanner are the same.

Treatment of domains with action costs depend on the SubPlanner (SP) and MainPlanner (MP) capabilities: (1) when both SP and MP handle action costs like CAP(FD/lama), CAP uses action costs during all stages. (2) When neither SP 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 Satisfying Track Settings

We evaluated CAP using three sets of instances: **(1) IPC2011 sequential satisfying (seq-sat) track instances**, **(2) large instances** of IPC2011 seq-sat track domains and assembly-mixed, and **(3) IPC2011 learning track, testing**

	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)	$\frac{c\left(\frac{\text{CAP}}{\text{(FD/lama)}}\right)}{c(\text{FD/lama})}$ 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
IPC2011 Sequencial Satisficing	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
Large Instances	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
IPC2011 Learning	barman-ipc11-learn(30)	0	30	30	-	4	29	28	.90±.03	6	28	28	.91±.04	0	0	0	-
	blocksworld-ipc11-learn(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-learn(30)	4	3	4	.97±.04	0	0	0	-	29	30	29	1.00±.09	0	0	0	-
	gripper-ipc11-learn(30)	0	0	0	-	0	0	0	-	0	0	0	-	0	0	0	-
	parking-ipc11-learn(30)	1	1	1	1.00±.00	14	9	10	1.00±.00	1	5	4	1.00±.00	0	0	0	-
	rover-ipc11-learn(30)	2	5	5	.99±.01	29	17	22	1.03±.08	18	16	18	1.06±.11	0	0	0	-
	satellite-ipc11-learn(30)	2	4	4	-	5	2	2	1.00±.00	0	0	0	-	0	0	0	-
	spanner-ipc11-learn(30)	0	0	0	-	0	0	0	-	0	0	0	-	0	0	0	-
	tpp-ipc11-learn(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: 30 min, 4 GB results for IPC2011 sequential satisfying track instances, large instances (see Table 3), and IPC2011 learning track test instances. Coverage results (#) and cost comparison for 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(31)	0	30	29	0	0	0	3	28	29	0	0	0	2	20	30	1	1	1
blocksworld-ipc11-learning(60)	6	5	6	0	0	0	26	24	26	18	0	27	19	19	19	20	18	30
depots-ipc11-learning(60)	4	3	3	1	2	4	0	0	0	3	3	3	30	30	30	30	30	30
gripper-ipc11-learning(56)	0	0	0	0	0	25	0	0	0	0	0	26	0	0	0	0	0	0
parking-ipc11-learning(37)	1	1	1	7	0	7	11	7	9	4	4	4	3	0	0	3	3	3
rover-ipc11-learning(59)	1	2	3	0	0	0	24	4	10	6	6	29	16	12	16	20	20	11
satellite-ipc11-learning(48)	1	4	4	0	0	7	1	0	0	2	2	18	0	0	0	0	0	0
spanner-ipc11-learning(30)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
tpp-ipc11-learning(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: IPC2011 learning track results. Coverages of our baselines, CAP, CAP with 7.5 preprocessing limit (pre7.5) (all with 15 min, 4 GB). Chrpa's baselines, Wizard and MUM are from (Chrpa, Vallati, and McCluskey 2014). Note that in (Chrpa, Vallati, and McCluskey 2014), MUM and Wizard were given 30 min (15 min for learning phase, 15 min for solving phase).

instances. All runs adhered to IPC seq-sat settings i.e. 30 min, 2GB memory limits. All phases of CAP (including preprocessing, main search and postprocessing) were included in these resource limitations.

We emphasize that **(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 are more difficult than **(1)**. However, the results on **(1)** are useful in order to address the two major concerns regarding 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, CAP(FD/h^{cea}) had significantly increased coverage compared to FD/h^{cea}. On the other hand, CAP(FD/lama) and CAP(Probe) both had worse coverage compared to FD/lama and Probe. Plan costs tend to be slightly higher when CAP is used. For example, $c(\text{CAP(FF)})/c(\text{FF})$, the ratio of plan costs of CAP(FF) to FF on problems solved by both 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. For **(3)**, we used the test instances of IPC2011 learning track. We reemphasize that this experiment was run with a 30 min time limit and 2 GB memory limit, unlike the IPC learning track setting which allows 15 min learning + 15 min planning + 4 GB memory.

Overall, CAP variants obtained significantly higher coverages. We see major improvements in barman, elevators, transport, woodworking, tpp. However, there is a severe degradation in openstacks, parking and visitall – the next subsection addresses how performance on these domains can be improved.

Improving CAP by Limiting the Preprocessing Time CAP is expected to perform well on domains that can be decomposed into serializable subproblems. On domains without this decomposable structure, CAP will not yield any benefits – all of 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 spent the majority (85-100%) of its time on preprocessing. While the preprocessing time varies, there

domain	IPC2011 Satisficing Track	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 Satisficing Track instances.

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 the 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 compare CAP performance with previous existing macro-based methods, we first compared CAP against Mar-

vin. Table 5 shows the coverages of FF, CAP(FF) and Marvin on **(1,2,3)**, with 30 min and 2 GB resource constraints. 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 used the unit-cost version of the problem. We also removed some domains because the latest 32bit binary of Marvin runs into segmentation faults or emits invalid plans on those domains. Interestingly, CAP(Marvin) and CAP(Marvin, pre15) improves upon Marvin on some domains, indicating that CAP is complementary to Marvin's plateau escaping macro.

	Domain	FF	CAP(FF)	CAP (FF, pre15)	Marvin	CAP (Marvin)	CAP (Marvin, pre15)
IPC2011 Sequencial Satisficing	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
Large Instances	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
IPC2011 Learning	barman-ipc11-learn(30)	0	30	30	1	30	30
	blocksworld-ipc11-learn(30)	6	6	6	12	10	11
	depots-ipc11-learn(30)	4	3	4	0	1	1
	satellite-ipc11-learn(30)	2	4	4	0	11	10
	tpp-ipc11-learn(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, woodworking, 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 bugs or PDDL incompatibilities (The FF family solved all domains successfully – see also the results in Table 1.)

Next, Table 2 compares CAP with the results of (Chrpa, Vallati, and McCluskey 2014). Experiments were run on the testing instances of IPC2011 learning track, with a 4 GB memory limit. While MUM and Wizard were given 15 min for learning *plus* 15 min for solving, CAP variants were given only the solving phase of 15 min, because unlike MUM and Wizard, CAP does not learn reusable domain knowledge. Also, CAP variants with a 7.5 min preprocessing time limit (included within the 15 min) are shown as pre7.5. All phases of CAP, including its preprocessing (macro generation) are included in the 15 min solving phase.

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 were run on a "3.0 GHz machine" (2014, p.5) while we used a 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 were verified using the Strathclyde VAL plan validation tool), while (Chrpa, Vallati, and McCluskey 2014) reported that their baseline FD/lama solved 46 instances.

Figure 2 shows that CAP and MUM improves performance on different domains – In particular, CAP performs well in barman and TPP, while MUM performs well in blocksworld, gripper, rover and satellite. Based on these results, CAP and MUM appear to be complementary, and their combination is an avenue for future work.

How Are CAP Macros Used?

So far, we have focused on the performance of CAP on benchmark instances. We now investigate the properties of CAP macros and their usage. First, we measured the length of all macros generated by the preprocessing phase, including unused macros, as well as the lengths of the macros that were actually used during the search. For example, Figure 3 clearly shows that CAP finds both long and short macros: There were > 1000 short macros of length 2, as well as at 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 (right side of Figure 3), the number of points decreases due to the exclusion of unused macros, but there is still a large number of long and short macros used. We observed a similar distribution for all other settings (benchmark sets and Sub/MainPlanner variations), but these are not shown due to space.

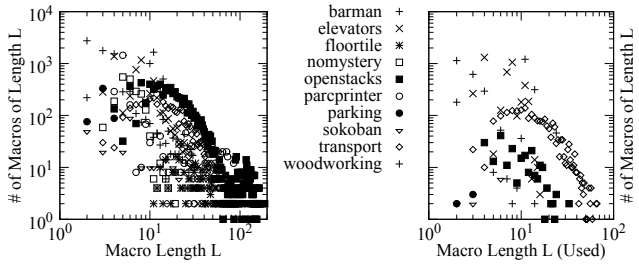

Figure 3: (Left) Distribution of the length of macros found by CAP(FD/lama) in IPC2011 seq-sat instances. (Right) Distribution of the length of macros *used in the solutions (plans)* for the same instances (same scales for both figures).

We also hypothesized that the large number of long macros generated by CAP does not harm the planner performance because the nullary ground macros generated by CAP have very limited applicability (although they are highly effective when applicable), and therefore, the effective branching factor does not increase significantly. We support these claims in Figure 4 (Left), which plots the # of node expansions by the baseline planner $X \in \{\text{FF, FD/lama, FD/}h_{\text{cea}}\text{, Probe}\}$ (all included) vs. # of nodes expanded by CAP(X). Each point represents an IPC2011 seq-

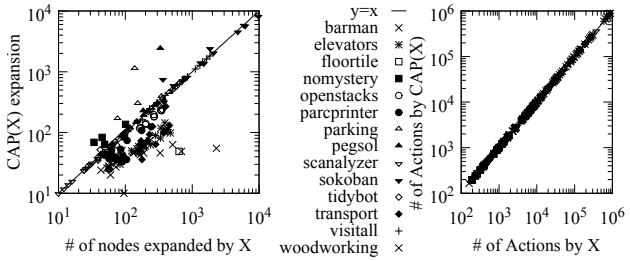

Figure 4: Each point represents IPC2011 seq-sat instances solved by baseline planner X and its CAP variant CAP(X). (Left) shows the number of nodes expanded where $X \in \{\text{FF, FD/lama, FD/}h_{\text{cea}}\text{, Probe}\}$. (Right) shows the number of instantiated ground actions where $X \in \{\text{FF, Probe}\}$.

sat instance solved by both X and CAP(X). While an increase in node expansion can be seen in a few domains, in most domains, the number of expansion by CAP(X) is the same or even *reduced* compared to X. This result shows that our nullary macros benefit planner performance even when they are rarely applicable. While previous macro methods were forced to impose macro length limitations in order to avoid the problem of generating many highly applicable, low-utility macros, CAP successfully takes the opposite approach of generating rarely applicable, high-utility macros.

Also, Figure 4 (Right), which plots the number of ground actions on the same set of instances solved by the baseline planner $X \in \{\text{FF, Probe}\}$ vs. that of the main search of CAP(X), shows that the increase in the number of ground actions due to CAP are negligible.

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 schedules 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.

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, each of which contains *a set of subgoals* and requires a significant amount of search by the SubPlanner. As shown in Table 1, CAP is complementary to Probe. CAP(Probe), which uses Probe as both the SubPlanner and MainPlanner, performs better than Probe by itself.

CAP macros can also be viewed as an extension to *T-macros* introduced in MORRIS (Minton 1985) along with *S-macros*. T-macros are used less frequently than S-macros are, but MORRIS benefits from T-macros when there are reoccuring sets of interacting subgoals.

CAP is an offline approach which statically analyzes a problem before the search. In contrast, Marvin (Coles and Smith 2007) embed macro learning into the search, and stratified planning (Chen, Xu, and Yao 2009) uses decomposition to focus the search online. An advantage of an offline approach is modularity. Different planners (including other macro-based methods) can be easily wrapped by an offline approach – indeed, we showed that CAP can even be used as a wrapper around Marvin.

Factored Planning (Amir and Engelhardt 2003; Brafman and Domshlak 2006) and CAP both seek to decompose problems in order to make them more tractable. The fundamental difference between Factored Planning and CAP is that Factored Planning seeks a complete decomposition of the problem (i.e., a partition) such that a bottom-up approach can be used to compose subplans into a complete plan. CAP, on the other hand, limits itself to seeking subproblems (components, as opposed to partitions) that may or may not be composable into a complete plan. While the current implementation of CAP uses Component Abstraction for decomposition, applying decomposition techniques investigated for Factored Planning (e.g., Causal Graph-based decomposition) is an interesting avenue for future work.

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 ground, nullary macros which are added to the domain. In contrast to previous macro systems that generate lifted macro operators, this has a significant benefit – since relatively few ground macros are added and instantiated, CAP avoids the macro utility problem without the need for macro filtering or bounds on macro length.

We showed that CAP can be used to find satisficing solutions to large problems 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 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 and avoids increasing the branching factor.

One promising direction for future work is exploitation of symmetry (Fox and Long 1998) among components. While macro reusability is completely optional in CAP, some subproblems may be very similar to each other when the component share the abstract type, and such reusability can be easily exploited by checking abstract type in order to further reduce the preprocessing time. Preliminary results with such a mechanism have been promising.

References

Amir, E., and Engelhardt, B. 2003. Factored planning. In *IJCAI*, volume 3, 929–935. Citeseer.

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.

Brafman, R. I., and Domshlak, C. 2006. Factored planning: How, when, and when not. In *AAAI*, volume 6, 809–814.

Chen, Y.; Xu, Y.; and Yao, G. 2009. Stratified planning. In *IJCAI*, 1665–1670.

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. 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.

Jonsson, A. 2007. The role of macros in tractable planning over causal graphs. In *IJCAI*, 1936–1941.

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*.

Minton, S. 1985. Selectively Generalizing Plans for Problem-Solving. In *International Joint Conference on Artificial Intelligence*, 596–599.

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.