Solving Large-Scale Planning Problems by Decomposition and Macro Generation

Submission 7

Abstract

Large-scale 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 and possibly generalized and reused to solve other subproblems. We show experimentally that our approach can be used to solve large problem instances with a repetitive structure (such as assembly planning) that are beyond the reach of current, state-of-the-art satisficing, classical 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 repetitive 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 100 instances of Widget A and 100 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, for a human, finding a satisficing solution for such an assembly problem is not difficult, as long as assembling single widget by itself is not difficult. 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 100 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 1 instance 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. 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 abstract subproblems where each subproblem corresponds to the assembly of a single type of product, like Widget A and Widget B.

Our approach uses the notion of "component abstractions" [Botea, Müller, and Schaeffer, 2004]. Botea et al made limited use of component abstractions, generating short (2-steps, in practice) macros which combines actions that affect the same single component at a time. 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 enhanced domain. This allows us to solve large problems by decomposing problems into independent subproblems and sequencing subsolutions to these subproblems.

Second, we identify *compatible groups* of component tasks that can be solved using the same plan (with parameter substitutions). This is a type of symmetry detection (c.f. [Fox and Long, 1998; Pochter, Zohar, and Rosenschein, 2011; Domshlak, Katz, and Shleyfman, 2013]). An advantage of this method is that it is conveniently implemented as part of the planner-independent wrapper, while symmetry breaking techniques are usually tightly coupled with specific search algorithms. In other words, this can add a (weak) symmetry-breaking technique to arbitrary underly-

ing domain-independent planner.

Third, in contrast to previous macro systems which tend to generate relatively short macro operators, our system generates large macros to solve entire tasks (e.g., building a complete widget). Our macro-operators are generated based on a particular initial state, and become increasingly difficult to apply as the search progresses and the current state diverges from the initial state. However, for the particular class of problems addressed by our system (mass-manufacturing type domains), it is natural to periodically reset the system to a stable, start point. Thus, we introduce *cyclic macros*, which addresses this issue of macro applicability by encoding an entire subtask *cycle*.

Fourth, our approach is completely planner independent and automated. 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. No modifications to the input domain/problem are necessary, and there is no need to provide additional training instances.

The resulting system, **CAP**, is capable of solving very large problems (with a decomposable structure) that are beyond the range of standard, state-of-the-art domain-independent planners. In our experiments, for example, CAP can solve Woodworking instances with 79 parts with 120% wood – in contrast, the largest standard IPC2011 instances of Woodworking has 13 parts, and standard, state-of-the-art planner Fast Downward can only solve 23 parts (with 120% wood). Even Probe, which can solve standard IPC Woodworking instances with only 31 expanded nodes [Lipovetzky and Geffner, 2011], can only solve 46 parts (with 140% wood, a weaker constraint than 120%).

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 (*component task*). Then we describe how we generate informative large macros, each corresponding to the result of solving each subproblem. We then describe how the evaluation of subproblems can be minimized by checking the plan-compatibility between subproblems. 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, we propose an approach to solving them by *automatically* decomposing them into independent subproblems and sequencing the solutions to them. An overview of our overall approach, **C**omponent **A**bstraction **P**lanner (CAP) is shown in Figure 1. CAP performs the following 5 steps:

1. *(Improved) Component Abstraction* (based on [Botea, Müller, and Schaeffer, 2004]): Perform a static analysis of the PDDL domain and problem in order to identify the independent components.
2. *Identifying Subproblems (component task)*: For each such component, extract the corresponding relevant portion of the initial and goal condition in the problem instance and form a component task.
3. *Solving Subproblems*: Solve the tasks while categorizing them according to *plan-compatibility*, which checks if a task can be solved with previously obtained plan and minimizes the redundant computation.
4. *Macro generation*: For each plan, generate a large macro operator.
5. Finally, solve the original problem with a new PDDL domain enhanced with macros. Then decode the macros in the result plan into primitive actions.

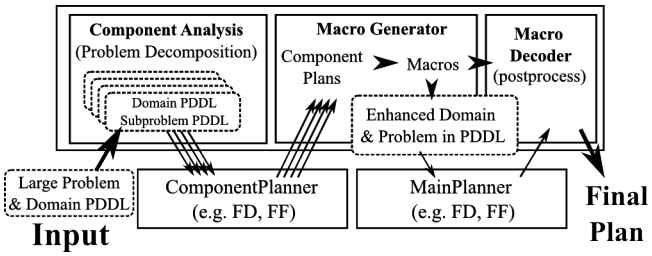

Figure 1: CAP System Overview. ComponentPlanner and MainPlanner are domain-independent planners, e.g., FD [Helmert, 2006], FF [Hoffmann and Nebel, 2001]. ComponentPlanner and MainPlanner can be the same planner, or different planners

Component Abstraction (based on [Botea, Müller, and Schaeffer, 2004])

We identify subproblems in large-scale, repetitive problems using an extended version of the component abstraction method by [Botea, Müller, and Schaeffer, 2004].

First, we build a *static graph* of the problem and cluster it into disjoint subgraphs called *abstract components*. This algorithm assumes typed PDDL domains (the types required for component abstraction can be automatically added to untyped domains using methods such as TIM [Fox and Long, 1998] or added straightforwardly by hand).

The static graph of a problem Π is an undirected graph $\langle V, E \rangle$, where nodes V are the objects in the problem and the edges E is a set of 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*.

Fig. 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 red" (allowed to be painted red somewhere in the plan), as opposed to, e.g., "temporarily painted" red (which can be modified by some actions like "paint" or "clean").

For such a static graph, the following is a Component Abstraction algorithm based on [Botea, Müller, and Schaeffer,

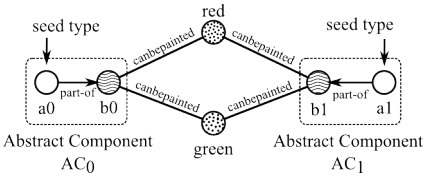

Figure 2: Example of a static graph and components

2004] with some modifications for our purpose. First, for each (ground) goal preposition g, 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 Fig. 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 $\{\mathsf{a0}, \mathsf{a1}\}$ and the components are $C = \{(\mathsf{a0}), (\mathsf{a1})\}$.]
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 there is an edge[1] (a, b) in the static graph, a belongs to the component ($a \in c$), and b does not belong to any components ($b \notin c, \forall c \in C$). [$f$ =(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. [$C = \{(\mathsf{a0}), (\mathsf{a1})\}$ are updated to $\{(\mathsf{a0\ b0}), (\mathsf{a1\ b1})\}$] 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 [Botea, Müller, and Schaeffer, 2004] is that we try all seed types s as long as some object of type s exist in the goals, while Botea et al. 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 Botea et al [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.

[1]An n-ary predicate $p(x_1, \ldots x_n)$ is treated as combinations of binary predicates $p_{ij}(x_i, x_j)(j \neq i)$.

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. Of a *fluent fringe fact* $f(a, b)$ of X, one parameter object (e.g. a) belongs to X and none of the other parameters (e.g. b) belong to any components. For each X, we collect all such facts in the initial and goal condition for *init* (X) and *goal* (X), respectively. For example, the initial state of the problem in Fig. 2 may contain a fact (not-painted b0), a fluent which takes $\mathsf{b0} \in AC_0$ 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 an *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:

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

Informally, O_X contains X, but does not contain the siblings of X, the components of the same abstract type. 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 local reasoning based on 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 problem domain. This enhanced domain is then solved using a standard domain-independent planner (MainPlanner in Fig 1). It is intended that the planner will use component macros to solve subproblems, inserting addi-

tional actions as necessary. If the resulting plan uses macro actions, 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: $\langle \textit{params}_1, \textit{pre}_1, e_1^+, e_1^-, c_1 \rangle$ and a_2: $\langle \textit{params}_2, \textit{pre}_2, e_2^+, e_2^-, c_2 \rangle$, where $\textit{params}_i$ is the parameters of action a_i, $\textit{pre}_i$ 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

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

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.

Cyclic Macros Component plans are generated by solving a component task, consisting of an abstract component, and its relevant parts of the initial state and goal. Thus, component macros, as described so far, are basically "shortcuts" which are applicable to a state which resembles the initial state, and achieve the associated subgoal in 1 step.

CAP is built on the assumption that we can often use these macros in a plan for the complete problem. For example, if we have 2 component macros m_a and m_b, we might want to use them in a plan such as: *Init* $\rightarrow$ $\textit{PrimitiveActions}_1$ $\rightarrow$ m_a $\rightarrow$ $\textit{PrimitiveActions}_2$ $\rightarrow$ m_b $\rightarrow$ *Done*. It is quite likely that in the initial state, the preconditions for both m_a and m_b are satisfied. However, as the search progresses and the state increasingly diverges from the initial state, it becomes increasingly unlikely that the preconditions for the macros are satisfied. Ideally, the planner can find sequences of primitive actions that set up the preconditions which allow the macros to be executed (e.g., $\textit{PrimitiveActions}_1$ establishes the preconditions for m_a, and $\textit{PrimitiveActions}_2$ establishes the preconditions for m_b). However, finding such action sequences may be quite difficult for the base level, domain-independent planner.

Consider an Elevator problem with 3 floors (Fig. 3): Assume that component macros m_1 and m_2 which transports one person from 1F to 2F and 3F, respectively, have been found. A forward heuristic search planner (currently the dominant approach in classical planning) will quickly decide to apply one of the macros (say, m_1). However, from the state that results after m_1 is executed, establishing the preconditions that allow the other macro (m_2) to be used can require a lot of search, and the planner can fail to successfully establish a chain of component macro executions.

Our approach to generating component macros so far has focused entirely on achieving the subgoals associated with each component task. This results in a set of macros where the 1st macro application is easy, but subsequent macro applications are difficult. To chain component macros together, we should seek macros that not only achieve the subgoals of their associated component task, but also seek to enable the

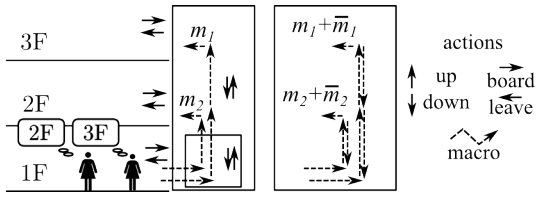

Figure 3: Cyclic macros in elevator domain.

application of other macros. One solution is to create macros which, in addition to achieving the "forward" objective of achieving the subgoals of a component task, also *restores* as much of the state as possible to the original state before its application. For example, in the Elevator example, the macro for moving one person between floors should move the person to the target floor (forward), and then move the elevator back to the first floor (restoration).

The "reverse" macros $\bar{m}_1, \bar{m}_2$ in Fig. 3 shortcuts in this direction. We solve a subproblem $\bar{\Pi}_{X,m_X}$ which restores the states, maintaining the previously achieved goals. When $m_X(I)$ is a result of applying m_X to I and $\textit{init}(X), \textit{goal}(X)$ are the fluent fringe facts of X in I, G;

$$\bar{\Pi}_{X,m_X} : \langle \mathcal{D}, O, m_X(I), \textit{goal}(X) \cup (I \setminus \textit{init}(X)) \rangle$$

If such a path is found, we merge m_X and $\bar{m}_X$ (using the merge operation (1) defined above) and use it as one unit, as shown in the right side of Fig. 3. This merged macro $m_X + \bar{m}_X = m_X^{\circlearrowleft}$ is called a *cyclic macro*. The time limit for seeking a reverse macro $\bar{m}_X$ is the same as for forward-macros (30 secs/component). In many domains, reverse macros may not exist. For example, in IPC Woodworking domain, assembling parts together to form a product consumes the parts (wood), and there is no way to restore consumable resources.

In all of the experiments below, CAP first seeks the forward macro m_X, then the reverse macro $\bar{m}_X$, and if $\bar{m}_X$ is found, CAP adds $m_X^{\circlearrowleft}$ to the enhanced domain and does not add m_X. However, if $\bar{m}_X$ is not found within the time limit, CAP adds the forward macro m_X to the enhanced domain.

Reusing Component Plans

The decomposition and macro generation method we proposed above allows us to solve large problems by identifying components, solving each component independently, and chaining the solutions to these subproblems (i.e., component macros) into a plan that solves the overall problem. This alone is a powerful technique which allows us to solve some difficult problems that can not be solved using current, domain-independent planners (see results below).

However, component abstraction and component planning can be made more efficient on very large problems with a *repetitive* structure. For example, suppose we have a problem which requires assembling 1000 instances of Widget A. Assuming that component abstraction correctly identifies the 1000 components corresponding to each instance of Widget A, ComponentPlanner must be called for each of the 1000 instances. Clearly, this is inefficient – a human would easily recognize that a plan for assembling 1 instance of Widget A can be reused when assembling 1000 instances.

Thus, CAP automatically identifies components which

are "compatible". Plan compatibility is defined as follows: Given two component problems $\Pi_X, \Pi_{X'}$ of abstract components X, X', respectively, if there exists a plan template P that (with the appropriate assignment of objects to the parameters of the actions in P) solves both Π_X and $\Pi_{X'}$, then Π_X and $\Pi_{X'}$ are compatible. Compatibility checking must find an appropriate mapping of objects between objects of X in P and the corresponding objects of X'. Existence of such a mapping between components in each subgroup can be guaranteed by first applying a simple pre-categorization that tests if they share the same abstract type, *init* (x) and *goal* (x) under parameter substitution $x = X, X'$.

Simply checking pairwise plan compatibility does not reduce the # of subproblems to be solved because the plans to be compared are available only after solving the subproblems. However, we can exploit the fact that compatibility is symmetric and transitive. First, when we check the compatibility between two components X, X', computing $P_{X'}$ is unnecessary due to symmetry. Also, by transitivity, we can check the compatibility between X' and yet another component X'' by checking the compatibility between X and X'', without having to compute $P_{X''}$.

Finally, to generate a macro operator that can be applied to any of the components in the compatibility group, we generalize the macro by *lifting* the ground plan. In principle, adding this lifted macro operator to the domain should be sufficient. However, current forward-search based planners tend to instantiate ground actions from action schema, and the process of instantiating the multitude of applicable ground instances of these macros (the overwhelming majority of which are useless) causes problems. Thus, instead of adding lifted macros to the domain, we generate fully grounded macro operators for each of the compatible components *for which these macros were generated in the first place*, thereby avoiding these issues (this may prevent some serendipitous usage of the lifted macros in some cases).

Experimental Results

In all experiments below, CAP is run on an Intel Xeon E5410@2.33GHz CPU with $30[\min], 2[\mathrm{GB}]$ memory limits. All phases of CAP (including preprocessing, executing MainPlanner on the enhanced domain, and postprocessing) are all included in these resource limitations.

As baselines, we evaluated FF (standard parameters), Probe, and 2 configurations of Fast Downward: (a) FD/LAMA2011 (iterated search), (b) FD/GBFS-CEA (Greedy Best First Search on context-enhanced additive heuristic).[2]

We tested the following CAP configurations: CAP(ff), CAP(lama), CAP(probe) and CAP(cea), which use FF, FD/LAMA2011, Probe, and FD/GBFS-CEA, respectively as both the ComponentPlanner (for component planning) and the MainPlanner (for solving the macro-enhanced problem generated by CAP).[3]

It is possible for ComponentPlanner and MainPlanner to be different planners (Fig. 1). Some planners may be better suited as the ComponentPlanner rather than the MainPlanner, and vice versa, so mixed configurations, e.g., CAP(ff+lama), which uses FF as ComponentPlanner and FD/LAMA2011 as MainPlanner, are possible, However, in this paper, for simplicity, we focus on configurations where ComponentPlanner and MainPlanner are the same.

Decomposition and Component Macros First we show that the problem decomposition (without compatibility checking) is effective. Below is a coverage result by CAP *without* compatibility checking, i.e. all subproblems are solved exhaustively, and macros are *not* generalized for compatibility groups.

The 4 baseline planners and 5 configurations of CAP were applied to 20 instances of the assembly-mixed, a modified version of the CELL-ASSEMBLY domain [Asai and Fukunaga, 2014] where problem #i is to assemble i instances of 2 different products. This resulted in the coverage results shown in Table 1 (# of problems solved out of 20).

Inspection of the plans generated show that subproblems are correctly identified and their solutions (captured as component macros) are sequenced to solve the overall problems. Thus, even without compatibility checking, CAP can outperform the baseline planners on some domains.

				CAP (no compatibility)				CAP			
ff	lama	probe	cea	(ff)	(lama)	(probe)	(cea)	(ff)	(lama)	(probe)	(cea)
3	3	4	2	18	15	19	18	20	20	20	20

Table 1: Coverage on CELL-ASSEMBLY domain for 4 baseline planners, 4 CAP configurations, and corresponding 4 variants of CAP without compatibility checking.

Compatibility Checking By removing unnecessary calls to ComponentPlanner, compatibility checking further improves performance, as shown in Table 1, which compares the coverage with and without compatibility checking on the same heterogeneous assembly-mixed instances as above. Compatibility checking improved coverage regardless of the ComponentPlanner/MainPlanner.

Evaluation on IPC2011 Instances We evaluate CAP using standard, IPC2011 instances (all of the sequential satisficing track STRIPS+action cost instances). *We emphasize that CAP is not intended to be applied to standard IPC instances – the motivation for CAP is scalability on decomposable problems that are much larger than standard IPC*

[2] It might seem that a possible, trivial baseline for comparison could be a modified standard planner, e.g., FF, that selects and plans for 1 "goal", then selects and plans for another goal, etc. However, selecting this "next goal" (whether "goal" means a single goal preposition, or a landmark, etc.) is, in general, nontrivial (consider the difficulty of this "baseline" for arbitrary IPC domains), and is in fact the crux of the problem being addressed by CAP (and also Probe, at a lower level of abstraction). Thus, we compare CAP to the base planners selected above.

[3] Treatment of domains with action costs depend on the ComponentPlanner (CP) and MainPlanner (MP) capabilities: [Case1] both the CP and the MP handle action costs (e.g., CAP(lama)) – CAP uses action costs during all stages. [Case2] Neither the CP nor MP handle action costs (e.g., CAP(ff)) – CAP removes action costs while planning, but in Table 3, plan costs are evaluated using the original domain definition (with costs).

instances. However, these results are useful in order to address the two major concerns of CAP when relatively small problems are given as input: (1) will the overhead of the preprocessing (component abstraction + macro generation) result in significantly decreased coverage compared to the base planner? (2) will solution quality suffer significantly?

The IPC2011 results in Table 3 shows that when CAP is applied to FF, (i.e., see the"ff", "CAP(ff)", and the adjacent column comparing the plan qualities "c(CAP(ff))$/c$(ff)"), coverage *increased* compared to FF (169 instances solved by FF, vs. 180 by CAP(ff)), while there is a slight penalty with respect to plan cost (the ratio of CAP(ff) plan cost to FF plan costs on problems solved by both is 1.18 on average with a standard deviation of 0.4). As would be expected, results vary per domain. Similarly, CAP(cea) has significantly increased coverage compared to FD/GBFS-CEA, with a slight cost penalty. CAP(probe) is comparable to Probe. On the other hand, CAP(lama) has both worse coverage and quality compared to FD/LAMA2011.

Evaluation on Large Instances

Table 3, which shows the results for very large instances, shows our main results. We evaluated CAP on assembly-mixed, the original motivating domain for this work, as well as large instances of generated using IPC-2011 instance generators for all IPC-2011 domains for which scalable problem generators could be obtained (we could not locate generators for pegsol,parcprinter, scanalyzer,sokoban).[4]

Note that although these large, IPC domain instances were generated using the same problem generators as the standard IPC-2011 instances (i.e., domains are unmodified), they are significantly larger than their standard, IPC counterparts (see Table 2 for a comparison).

domain	large instances	IPC'11 instances
woodworking	34-81 parts,1.2-1.4 woods	3-13 parts,1.2-1.4 woods
visitall	61-80 grids	12-50 grids
transport	66 nodes, 20-60 packages	50-66 nodes,20-22 packages
openstacks	170-290 stacks	50-250 stacks
elevators	40 floors 62-137 passengers	16-40 floors 13-60 passengers
barman	47-78 shots, 47-71 cocktails	10-15 shots, 8-11 cocktails
tidybot	144-169 area, 8-20 goals	81-144 area, 5-15 goals
parking	15-18 curbs 28-34 cars	10-15 curbs 20-30 cars
assembly-mixed	1-20 product A, 1-20 product B	n/a (not an IPC domain)

Table 2: Large instances: some characteristics compared to Standard IPC instances

When/How Does CAP fail? CAP is designed for 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, for most of the large domains, CAP spends the majority (in many cases, 99-100%) of its time on preprocessing.* While the time spent preprocessing depends on the domain characteristics, there are essentially 4 cases: 1) There are few, if any, component problems so preprocessing is fast (e.g. tidybot); 2) Most component problems are compatible (e.g. assembly-mixed) so preprocessing is fast; 3) Although useful macros are discovered, the preprocessing consumes too much time, leaving insufficient time for the MainPlanner to succeed (e.g. elevators); or 4) preprocessing is effort is wasted because the macros are not useful to the MainPlanner (e.g. openstacks).

Improving CAP By Limiting Preprocessing Time A simple approach to alleviate failures due to cases 3 and 4 above is to limit the time CAP spends on its macro generation phase. As shown in Table 3, running CAP with a 15 minute preprocessing limit (50% of the 30-minute overall limit) dramatically improves coverage for all of the CAP variants. 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 3). 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 4).

Domain	CAP ff mean±sd	CAP lama mean±sd	CAP probe mean±sd	CAP cea mean±sd
assembly-mixed-large(20)	.80±.04	.04±.04	.76±.04	.77±.03
barman-large(20)	.99±.00	1.00±.00	1.00±.00	1.00±.00
elevators-large(20)	.99±.00	.90±.30	.99±.02	1.00±.00
openstacks-large(21)	.96±.08	.90±.29	1.00±.00	1.00±.00
parking(20)	.58±.48	.50±.46	.92±.02	.56±.43
tidybot-large(20)	.14±.30	.00±.00	.03±.06	.02±.02
transport-large(20)	.98±.01	.77±.31	.89±.16	.84±.19
visitall-large(20)	1.00±.00	.45±.50	.95±.22	.95±.22
woodworking-large(20)	.50±.18	.95±.12	.53±.26	.99±.02
Average	.77±.35	.61±.46	.79±.33	.79±.35

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

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 only on identifying efficient cyclic solutions for mass manufacturing multiple instances of one particular product. In CAP terminology, ACP assumes that its input PDDL domain and problem files corresponds to exactly 1 "component" and nothing else. Thus, ACP can not handle *heterogeneous*, repetitive problems, e.g., a manufacturing order to assemble n_1 instances of product p_1 and n_2 instances of product p_2. Since ACP does not perform problem decomposition, ACP can not even be directly applied to homogeneous problems where a repeated

[4]However, extrapolating from the Table 3 results on the standard IPC-2011 instances of pegsol, parcprinter, scanalyzer, sokoban, it seems safe to say that the overall trends would not be significantly affected even if large instances of these domains were available.

Domain $c(x)$:plan cost #:coverage	ff # mean±sd	CAP ff # $\frac{c(\text{CAP ff})}{c(\text{ff})}$ mean±sd	lama # mean±sd	CAP lama # $\frac{c(\text{CAP lama})}{c(\text{lama})}$ mean±sd	probe # mean±sd	CAP probe # $\frac{c(\text{CAP probe})}{c(\text{probe})}$ mean±sd	cea # mean±sd	CAP cea # $\frac{c(\text{CAP cea})}{c(\text{cea})}$ mean±sd
barman-ipc11(20)	0	20 -	20	20 .86±.06	20	20 1.13±.15	0	16 -
elevators-ipc11(20)	19	20 2.11±.55	19	17 1.45±.49	17	16 1.56±.23	3	18 1.42±.10
floortile-ipc11(20)	10	4 .99±.03	4	5 1.17±.20	5	3 .92±.06	5	5 .98±.05
nomystery-ipc11(20)	8	6 .98±.11	12	7 .99±.01	6	6 .98±.04	6	6 1.04±.05
openstacks-ipc11(20)	18	0 -	15	3 .93±.06	11	8 1.02±.14	0	0 -
parcprinter-ipc11(20)	20	18 .99±.03	14	17 .98±.08	14	18 .92±.09	14	13 1.00±.00
parking-ipc11(20)	5	6 .99±.01	16	15 1.12±.28	10	8 1.12±.22	8	9 1.00±.00
pegsol-ipc11(20)	20	20 1.00±.00	20	18 1.00±.00	20	20 1.00±.00	20	20 1.00±.00
scanalyzer-ipc11(20)	19	19 1.00±.00	18	18 1.01±.04	17	17 1.00±.01	17	17 1.00±.00
sokoban-ipc11(20)	15	15 1.02±.08	18	18 1.02±.26	14	16 1.00±.03	12	13 1.05±.20
tidybot-ipc11(20)	13	13 1.00±.00	13	14 1.10±.24	16	17 1.00±.00	18	17 1.00±.00
transport-ipc11(20)	8	16 1.41±.30	17	15 1.60±.35	13	13 1.36±.20	6	13 1.83±.27
visitall-ipc11(20)	4	4 1.00±.00	20	10 1.00±.01	16	15 1.01±.02	3	3 1.00±.00
woodworking-ipc11(20)	10	19 1.00±.03	17	15 1.02±.03	20	18 1.03±.04	18	20 1.15±.17
Summary	169	180 1.18±.44	223	192 1.09±.31	199	195 1.08±.21	130	170 1.07±.21

Domain $c(x)$:plan cost #:coverage	ff # mean±sd	CAP ff # $\frac{c(\text{CAP ff pre15})}{c(\text{ff})}$ mean±sd	CAP ff pre15 # mean±sd	lama # mean±sd	CAP lama # $\frac{c(\text{CAP lama pre15})}{c(\text{lama})}$ mean±sd	CAP lama pre15 # mean±sd	probe # mean±sd	CAP probe # $\frac{c(\text{CAP probe pre15})}{c(\text{probe})}$ mean±sd	CAP probe pre15 # mean±sd	cea # mean±sd	CAP cea # $\frac{c(\text{CAP cea pre15})}{c(\text{cea})}$ mean±sd	CAP cea pre15 # mean±sd
assembly-mixed-large(20)	3	20	20 .76±.09	3	20	18 .98±.08	4	20	20 .72±.18	2	20	20 .54±.18
barman-large(20)	0	9	8 -	5	0	17 .96±.01	2	0	16 .49±.10	0	0	0 -
elevators-large(20)	4	7	7 2.43±.55	9	0	15 1.70±.22	0	0	10 -	0	1	0 -
openstacks-large(21)	14	0	0 -	18	0	16 1.00±.02	0	0	0 -	0	0	0 -
parking(20)	2	1	1 1.00±.00	20	10	8 .89±.18	0	0	0 -	3	3	2 1.00±.00
tidybot-large(20)	10	8	10 1.00±.00	6	9	9 1.09±.19	11	9	20 .86±.31	7	7	7 1.00±.00
transport-large(20)	0	8	9 -	3	5	11 1.48±.35	4	4	4 1.20±.07	0	0	0 -
visitall-large(20)	0	0	0 -	13	0	15 1.00±.00	0	0	0 -	0	0	0 -
woodworking-large(20)	3	0	0 -	0	3	6 -	3	3	20 1.01±.01	0	7	13 -
Summary	36	53	55 1.28±.68	77	47	**115** 1.10±.30	24	36	90 .87±.30	12	38	42 .92±.19

Table 3: Coverage results (#) and cost comparison on IPC'11 instances and Large instances, with 4 baseline planners and CAP with 4 combinations of those planners, and also their variants with 15[min] preprocessing time limit("pre15"). Cost ratios include only instances that were solved by both configurations. E.g., on the barman-large domain, FD/LAMA2011 solved 5 instances, CAP(lama) solved 0 instances, CAP(lama, pre15) with a 15-min preprocessing limit solved 17 instances, and the average cost ratio c(CAP(lama, pre15))/c(lama) = .96 on the 5 instances solved by both FD/LAMA2011 and CAP(lama, pre15).

structure is embedded as part of a large, complex problem. CAP does not have these limitations, and is therefore a more general approach.

Component abstraction in CAP is based on the algorithm originally introduced by [Botea, Müller, and Schaeffer, 2004] for CA-ED. CAP differs fundamentally from CA-ED in the way that the component abstraction is used for macro generation. CA-ED only uses AC's in a (brute-force) enumeration of short action sequences (macros) relevant to the AC's. Most such macros are useless (filtered by CA-ED), and the macros used by CA-ED are short (2-steps). In contrast, CAP is **goal-driven**. We extract subgoals relevant to AC's, and solve subproblems corresponding to the AC's + their related subgoals, and plans for these subgoals are generalized into macros. This results in macros that are more powerful than the short macro fragments used by CA-ED. Note that CAP does not apply any macro filtering methods, in contrast to CA-ED.

The problem decomposition approach in CAP is related to symmetry-based approaches. An early version of Marvin (Release 1, [Coles and Smith, 2004]) generated macros using *instance reduction* (IR), which extracts groups of subproblems that are compatible according to an almost-symmetry criterion (an extension of the symmetry criterion of [Fox and Long, 1999]). IR identifies objects that are almost-symmetric, chooses one examplar from each group of related objects, and extracts a subproblem using the "predicates whose entities are wholly contained within this set of exemplars". Consider a problem that requires building a bicycle and an automobile. Component abstraction can identify the bicycle component and automobile component. However, since assembling the bicycle and automobile require completely different parts and procedures, IR would not decompose these two subtasks. In general, symmetry-driven IR and component abstraction are orthogonal. Components are not necessarily symmetric (e.g., bicycles vs automobiles), and component abstraction by itself does not identify "similar"/symmetric components.

Unlike symmetry-based macros generated by IR, CAP macros are not necessarily intended to be reusable. Reusability is not necessary for a macro to be useful – providing 1-step solutions for subproblems that can be sequenced by MainPlanner is by itself quite powerful. While our "plan compatibility" (PC) criterion captures one type of symme-

try in order to avoid wasted calls to the ComponentPlanner, CAP can function moderately well on some domains even with PC disabled (see the assembly-mixed results in the "Decomposition and Component Macros" subsection).

Another significant difference between IR and CAP is the generation of cyclic macros in CAP (in our terminology, IR only generates forward macros). In some domains, forward macros alone can not be sequenced together into a complete solution, and reverse macros that restore an initial state so that the forward macros can be applied are necessary.

Compared to the other symmetry-breaking techniques by [Pochter, Zohar, and Rosenschein, 2011] and [Domshlak, Katz, and Shleyfman, 2013], an advantage of PC is that it was conveniently implemented as part of the planner-independent wrapper, while symmetry breaking techniques are usually tightly coupled with specific search algorithms. In other words, PC can add a rather simple symmetry-breaking ability to arbitrary domain-independent planner.

CAP can be viewed as a method for identifying and connecting a tractable set of waypoints in the search space. In that sense, it is somewhat related to Probe [Lipovetzky and Geffner, 2011], a best-first search augmented by "probes" that try to reach a goal from the current search state by quickly traversing a sequence of subgoals that lead to the goal. CAP and Probe work at different levels of abstraction: Probe seeks the next unachieved landmark in a "consistent" greedy chain, according to a consistency notion based on [Lipovetzky and Geffner, 2009]. CAP identifies serializable subproblems (Component Problems), solves them with the ComponentPlanner, and tries to chain the solutions using the MainPlanner. These component problems do *not* correspond to neighboring landmarks – each component problem may be comparable to an entire (relatively easy) IPC benchmark problem, requiring a significant amount of search by the ComponentPlanner. As shown in Table 3, 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 on our large instances (Table 3).

A widely used macro learning technique is the use of small "training instances" which are solved and analyzed in order to extract useful macros in systems such as SOL-EP [Botea, Müller, and Schaeffer, 2004], WIZARD [Newton et al., 2007], and [Chrpa, 2010]. For problems with a repetitive structure, our component analysis approach can be seen as a way to completely *automatically* extract and analyze such "training instances" from the problem 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 be implemented as preprocessors, as is our current implementation of CAP, and call state-of-the-art planners internally (Fig. 1). 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.

Conclusions

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

We showed that a fully automated, decomposition approach of generating component macros based on solutions to component tasks 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 completely automates a decomposition-based approach to solving large, decomposable problems, and is implemented as a preprocessor/wrapper for standard domain-independent classical planners. Our experimental results show that CAP can be successfully applied to 4 different combinations of planners (using different heuristics). To minimize the number of calls to the component planner, compatibility groups are used in order to identify groups of components that can be solved using the same plan template, enabling the reuse of component macros and resulting in significant speedups.

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 (Fig. 1). Preliminary experiments show that using a fast, greedy planner (e.g., FF) as the ComponentPlanner and a more deliberative planner as MainPlanner (e.g., FD/LAMA2011) 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 *ICAPS*.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Using component abstraction for automatic generation of macro-actions. In *ICAPS*, 181–190.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012. Optimizing plans through analysis of action dependencies and independencies. In *ICAPS*.

Chrpa, L. 2010. Generation of macro-operators via investigation of action dependencies in plans. *Knowledge Engineering Review* 25(3):281.

Coles, A., and Smith, A. 2004. Marvin: Macro actions from reduced versions of the instance (extended abstract). In *International Planning Contest (IPC-4) Proceedings*, 24–26. http://www.tzi.de/ edelkamp/ipc-4/IPC-4.pdf.

Coles, A., and Smith, A. 2007. Marvin: A heuristic search planner with online macro-action learning. *J. Artif. Intell. Res. (JAIR)* 28:119–156.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2013. Symmetry breaking: Satisficing planning and landmark heuristics. In *ICAPS*.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *J. Artif. Intell. Res.(JAIR)*.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proc. IJCAI*, 956–961.

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.

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.

Newton, M. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning macro-actions for arbitrary planners and domains. In *ICAPS*, 256–263.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting problem symmetries in state-based planners. In *AAAI*.