Solving Large-Scale Planning Problems by Decomposition and Macro Generation

Submission 1534

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 the performance of 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 easier 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 WidgetA and 100 instances of WidgetB. 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, particularly when assembly a single instance of a widget by itself is easy. It is clear that this problem consists of easily serializable subgoals [Korf, 1987]. The obvious strategy is to find a plan to assemble 1 instance of Widget A and a plan to assemble 1 instance of Widget B, and then find some way to combine these building blocks in order to assemble 100 instances of both widgets.

In this paper, we propose an approach to solving such classes of planning problems by automatic decomposition into easier subproblems. For example, in a *heterogeneous* CELL-ASSEMBLY domain where we are given a large number of parts that must be assembled into a set of products, our system extracts a set of abstract subproblems where each subproblem corresponds to the assembly of a single type of product.

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 combined actions that affected objects that were related according to the abstraction. We take the idea of component abstraction much further: First, we propose a method of identifying component tasks that consist of an abstract component, as well as the initial and goal propositions that are relevant to that component and generate large macros that solve an entire subproblem. These *component macros* are added to the domain, and a standard domain-independent planner is used to solve the satisficing problem for the enhanced domain. This allows us to solve large problems by decomposing problems into independent subproblems and sequencing subsolutions to these subproblems. Second, we propose methods to identify *compatible groups* of component tasks that can be solved using the same plan (with parameter substitutions).

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 per-

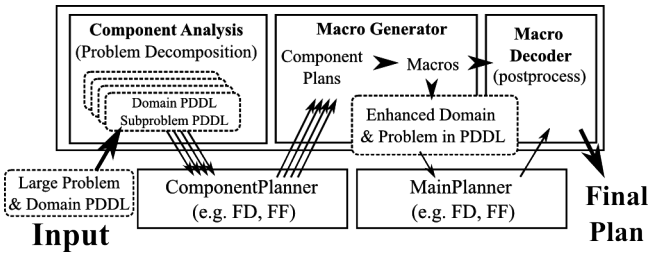

Figure 1: CAP System Overview. ComponentPlanner and MainPlanner are domain-independent planners, e.g., Fast Downward [Helmert, 2006], FF [Nebel, 2001]. ComponentPlanner and MainPlanner can be the same planner, or different planners

forms 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.

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, but this is not restrictive, since 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. 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").

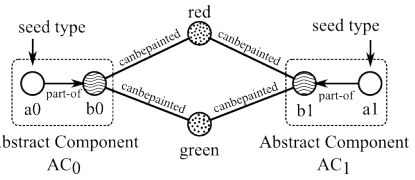

Figure 2: Example of a static graph and components

For such a static graph, the following is a Component Abstraction algorithm based on [Botea, Müller, and Schaeffer, 2004] with some modifications for our purpose. First, for each (ground) goal preposition g, we collect the types of their parameters *params* (g) into a queue T. We then iteratively run the following procedure (text in brackets "[]" describes the corresponding behavior in Fig. 2):

1. Pop one *seed type* s from T [○ is selected as s].
2. For each object (node) o of type s, initialize (single-node) abstract components and store them in a set C. [The seed objects are $\{$a0, a1$\}$ and the components are $C = \{$(a0), (a1)$\}$.]
3. Select one *fringe* predicate p of a component. A *fringe* predicate is an edge (a, b), such that a belongs to some component, and b does not belong to any component. [p =(part-of a0 b0).] Mark the name of p ["part-of"] as *tried* and do not select it twice. If no such p exists, append the current C to the final results R, reset $C = \emptyset$, then return to step 1. If T is also exhausted in step 1, return R.
4. If p exists, collect all fringe predicates of the same name as p. [(part-of a0 b0) and (part-of a1 b1).]
5. Extend the components by merging each with the arguments in the selected predicates. [$C = \{$(a0), (a1)$\}$ are updated to $\{$(a0 b0), (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. Another difference is that 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.

For an abstract component AC, we can define a *component task* as a triple of (1) AC, (2) a subset of propositions from the initial states relevant to AC, and (3) a subset of goal propositions relevant to AC. In order to find (2) and (3), we collect the *fluent facts*, the facts which are not static. Since the facts directly relevant to AC are its fringes, we collect all fluent fringe facts of AC in the initial and goal condition for (1) and (2), respectively.

For example, the initial state of the problem in Fig. 2 may include a fact (not-painted b0), which is in the fringe of AC_0 and is fluent because it can be removed by some actions, such as (paint ?b). Similarly, facts like (being b0 red) are the possible candidates of the 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*. Now let's see how a component problem can be expanded from a component task.

Let $X = \{o_0, o_1 \ldots\}$ 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:

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

Informally, O_X contains X, but does not contain the siblings of X. Note that E_X means *environment objects* that are not part of any components. In I_X, any ill-defined propositions (containing removed objects) are cleaned up. G_X is same as the goal conditions of the component task of X.

We solve this component problem Π_X with a domain-independent planner such as Fast Downward [Helmert, 2006] or Fast Forward [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 additional 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: $\langle \mathit{params}_1, \mathit{pre}_1, e_1^+, e_1^-, c_1 \rangle$ and a_2: $\langle \mathit{params}_2, \mathit{pre}_2, e_2^+, e_2^-, c_2 \rangle$, where params_i is the parameters of action a_i, pre_i are the preconditions of a_i, e_i^+ and e_i^- are the add and delete effects of a_i, and c_i is the cost of a_i. A merged action equivalent to sequentially executing a_1 and a_2 is defined as

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

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

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* $\rightarrow$ $\mathit{PrimitiveActions}_1$ $\rightarrow$ m_a $\rightarrow$ $\mathit{PrimitiveActions}_2$ $\rightarrow$ 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., $\mathit{PrimitiveActions}_1$ establishes the preconditions for m_a, and $\mathit{PrimitiveActions}_2$ 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 the first floor to the second and third floors, 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 compo-

nent macro executions.

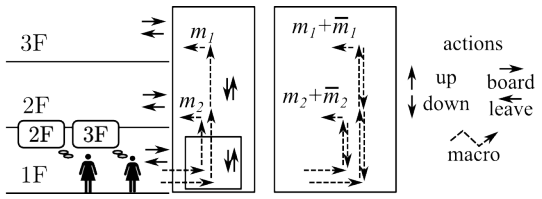

Figure 3: Cyclic macros in elevator domain.

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 can be difficult. If we intend component macros to be chained together, we should seek macros that not only achieve the subgoals of their associated component task, but also seek to enable the application of other macros. One solution is to create component 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 are macros which shortcuts in this direction. We solve a subproblem $\bar{\Pi}_{X,m_X}$ which restore the states, while maintaining the previously achieved goals. $\bar{\Pi}_{X,m_X}$ is

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

If such a path is found, we merge m_X and $\bar{m}_X$ (following the operation in (1)) 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 seconds per component). In many domains, it is possible that reverse macros do not exist. For example, in some assembly domains such as the 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. 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 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 by far more efficient when confronted with very large problems with a repetitive structure. For example, suppose we have a problem which requires the assembly of 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.

CAP implements this feature by automatically identifying 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 and the same fringe nodes.

In the worst case, simply checking the pairwise plan compatibility still requires the enumeration of all component plans that should be compared to each other, which means all subproblems still needs to be solved. However, we can exploit the fact that compatibility is a symmetric and transitive relationship. First, when we check the compatibility between two components X, X', we no longer have to compute $P_{X'}$ because checking if P_X applies to $\Pi_{X'}$ suffices. 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 IPC settings ($30[\mathrm{min}], 2[\mathrm{GB}]$ memory). All phases of CAP (including preprocessing, executing MainPlanner on the enhanced domain, and postprocessing) are all accounted for in these resource limitations.

As baselines, we evaluated FF (standard parameters) and 2 configurations of Fast Downward: (a) FD/LAMA2011 (b) FD/CEA (Context-enhanced additive heuristic). We tested

the following configurations of CAP, where ComponentPlanner is the planner used for component planning and MainPlanner is the planner used to solve the enhanced problem (which includes ground macros generated by CAP).

	ComponentPlanner	MainPlanner
CAP(ff+ff)	FF	FF
CAP(lama+lama)	FD/LAMA2011	FD/LAMA2011
CAP(cea+cea)	FD/h_{cea}	FD/h_{cea}
CAP(ff+lama)	FF	FD/LAMA2011

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

The 3 baseline planners and 4 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 following coverage results (# of problems solved out of 20).

Baseline			CAP			
cea	lama	ff	(cea+cea)	(lama+lama)	(ff+ff)	(ff+lama)
2	4	5	**20**	**20**	**20**	**20**

The CAP configurations clearly outperformed the baseline planners. 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.

Compatibility Checking By removing unnecessary calls to ComponentPlanner, compatibility checking further improves performance as in Fig. 4, which compares the runtimes with and without compatibility checking on the same heterogeneous CELL-ASSEMBLY instances as above. Compatibility checking reduces runtime by a factor of 2-10 (note that both axes are logarithmic scale), regardless of ComponentPlanner/MainPlanner.

Overall Evaluation Table 1 shows the performance of the 4 CAP variants on a set of large problems.

	baseline	CAP	baseline	CAP	baseline	CAP	CAP
domains	ff	(ff+ff)	lama	(lama+lama)	cea	(cea+cea)	(ff+lama)
barman-L(20)	0	**20**	4	4	0	6	**20**
assembly-mixed(20)	4	**20**	4	**20**	2	**20**	**20**
elevator-L(20)	6	**20**	7	4	0	4	**20**
rovers-L(20)	0	0	7	2	0	0	**10**
satellite-L(20)	7	**14**	6	8	0	6	9
woodworking-L(90)	9	13	24	29	16	23	**67**

Table 1: Coverage results of 30 min (including all pre/postprocessing) experiments. Best coverage results are in boldface. Underlines are CAP results better than its corresponding (non-CAP) baseline planners. "lama" = FD/LAMA2011 and "cea" = FD with Lazy GBFS and h_{cea}.

Most of these problems were generated by IPC generator (woodworking, rovers, floortile, barman, satellite). Types (based on unary predicates) were manually added to satellite instances. Assembly-mixed and Elevator instances were generated using our custom scripts. To make the domains compatible with FF, we manually removed :action-costs and :numeric-fluents from all instances[1].

Figure 1 shows that in most domains, the CAP(x+x) planner significantly outperforms the baseline planner x for all $x \in \{$FF, FD/LAMA2011, FD/$h_{cea}$$)\}$. This shows that the success of CAP is not dependent on any particular planner (or heuristic). Figure 1 also shows that CAP(FF, FD/LAMA2011) performed very well overall. This is because FF solves easy problems (such as component planning problems) quickly, and is well-suited as ComponentPlanner, while FD/LAMA2011 is better suited for solving hard problems, and is well-suited as MainPlanner, the planner used to solve the enhanced problem.

CAP is designed for domains that have serializable subgoals that can be decomposed relatively easily. 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. However, in such cases, the macro-generation step finishes very quickly because there are few, if any, component problems to be solved by ComponentPlanner. which is the most time-consuming part of the macro generation. Nevertheless, there may be domains and usage cases where time wasted by CAP on non-decomposable domains is an issue. In preliminary experiments using the same domains as Table 1), we have found that running the CAP macro generator with a time limit of 5-15 minutes is sufficient to find most of the useful macros that are found in a full run of the CAP preprocessor. Thus, for arbitrary domains where it is unknown whether decomposable, serializable subgoals exists, running the CAP preprocessor with a time limit is a promising approach to avoiding wasting too much time on fruitless runs of the CAP preprocessor.

In addition to the large instances in Table 1, we also tested IPC benchmark instances of other domains that we believed would be amenable to decomposition, including airport-strips, driverslog, TPP, depot, logistics, myconic, mystery, zenotravel, gripper, childsnack, floortile, sokoban-sat11-strips, pegsol, scanalyzer, pipesworld-tankage/notankage. For each problem, we manually added types if it was typeless. Unfortunately, most of the standard IPC benchmark problems are too easy to evaluate the effectiveness of our approach (FD/LAMA finds their first solution within 30 minutes). However we note that in gripper, airport, driverslog and TPP domain, CAP reliably finds component macros and compatible components. On the other hand, zenotravel, pipesworld-tankage/notankage, childsnack, and floortile could not be decomposed by CAP.

Search Behavior Fig. 5 shows how the macro compress/shortcuts the state space for MainPlanner. Since search depth is closely correlated with plan length, Fig. 5 shows that our approach has dramatically reduced the amount of search

[1] All test domains are at:
https://github.com/aaai15submission1534/domains

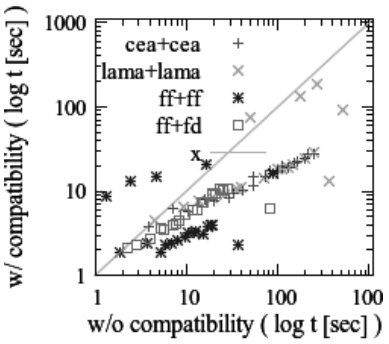

Figure 4: Total run time (logarithmic) of CAP on assembly-mixed with and without compatibility.

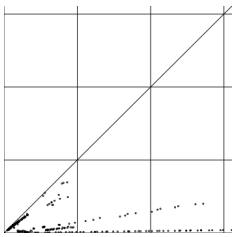

Figure 5: Plot of actual plan length (horizontal) vs the plan length before decoding macros (vertical) on all solved instances in Table 1. One grid tic corresponds to 1000 plan steps (unit cost).

performed by the MainPlanner on the solved instances (typically by at least a factor of 5, and up to hundreds).

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 structure is embedded as part of a large, complex problem. CAP does not have these limitations, and is therefore a more general approach. On the other hand, ACP implements a search algorithm which specifically seeks to find an efficient (minimal makespan) cyclic plan, so on problems where both methods are applicable, and combined with a postprocessing scheduler, ACP may generate more efficient parallel plans than CAP. Integrating the makespan optimization techniques of ACP into CAP is an avenue for future work.

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 that achieve 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.

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 instances and its domain and generate an enhanced domain. In contrast, on-line 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. Unlike previous approaches which required hand-picked training instances, 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 approach can be successfully applied to 4 different combinations of planners (using different heuristics). In order 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.

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

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

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.

Korf, R. E. 1987. Planning as search: A quantitative approach. *Artificial Intelligence* 33(1):65–88.

Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

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