Applying Problem Decomposition to Extremely Large Domains

Masataro Asai and **Alex Fukunaga**
Department of General Systems Studies
Graduate School of Arts and Sciences
The University of Tokyo

Abstract

Despite the great improvement in the existing planning technology, their ability is still limited if we try to solve extremely large domains like $N > 1000$. We propose a method that try to reduce the problem size by decomposing a problem into a set of cyclic planning problems. We categorized the objects in a problem into groups, using Abstract Type and Abstract Task. We then solve each subproblem per group as a cyclic planning problem, which can be solved efficiently with Steady State abstraction.

1 Introduction

Recent improvements in classical planning allowed planners to solve larger and larger domains by modeling planning as satisfiability and as heuristic search.

However even this is not enough for large-scale problems such as manufacturing domains which requires hundreds or thousands of objects to be processed at once. Since STRIPS planning is PSPACE-complete[Bylander, 1994] , it is impractical to directly solve such large problems with existing, search-based approaches.

A recent study [Ochi et al., 2013] showed that although standard domain-independent planners were capable of generating plans for assembling a single instance of a complex product, generating plans for assembling multiple instances of a product was quite challenging. For example, generating plans to assemble 4-6 instances of a relatively simple product in a 2-arm cell assembly system pushed the limits of State-of-the-Art domain-independent planners. However, real-world CELL-ASSEMBLY applications require mass production of hundreds/thousands of instances of a product.

As another example, consider the standard IPC benchmark Elevator domain, where the task is to efficiently transport people between floors using several elevators. In the IPC benchmark instances, the number of passengers is around 7 times the number of elevators in the satisficing track, and in the optimization track, the number of passengers is comparable to the number of elevators. However, (as we demonstrate in Sec. 4) a crowded elevator scenario where the number of passengers is more than 40 times the number of elevators is beyond the capabilities of state-of-the-art planners.

These are examples of domains where the problem instances consist of sets of tasks that are basically independent and decomposable, particularly if there is no optimality requirement. In the cell assembly domain, constructing a single product is relatively easy. In the elevator domain, transporting a single passenger from its initial location to its destination is easy. While standard, search-based planners struggle to find solutions to large-scale instances of these domains, a human can easily come up with satisficing plans for these problems by decomposing the problem.

In this paper, we propose an approach to decomposing large-scale problems by identifying groups of easy subproblems. In a 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. Similarly, in a large-scale elevator domain, our system extracts a set of abstract subproblems where each subproblem corresponds to loading N (generic) people into an elevator on a particular floor F_I and moving them to their particular goal floor F_G. In domains such as cell assembly and elevator where there are no resource constraints, the original, large-scale problems can be solved by sequencing solutions to the subproblems.

Our approach extends the previous work which identifies "component abstractions" for the purpose of finding macro-operators [Botea, Müller, and Schaeffer, 2004]. We identify abstract tasks that consist of an abstract component, plus the initial and goal propositions that are relevant to that component. We show experimentally that this approach is capable of decomposing large-scale, CELL-ASSEMBLY problems into a batch of orders that can be solved by a cyclic planner such as the recently proposed system by [Asai and Fukunaga, 2014]. We also show that our system can be used to decompose large-scale versions of some ICAPS benchmark domains (Elevator, Woodworking, Rover, Barman) into more tractable subproblems.

The rest of the paper is organized as follows. First, we explain our overall approach to decomposition-based solution of large-scale problems (Sec. 2). We then explain the limitation of existing cyclic planning model further. Next we de-

scribe the notion of abstract type and *Abstract Component* originally came from [2004] and its extension (*Attribute*) in Sec. 3.1. Then we describe the notion of abstract task in Sec. 3.2. Finally, we show experimental decomposition results of extremely large PDDL domains, including both large CELL-ASSEMBLY problems as well as very large instances of standard benchmark domains.

2 Background and Motivation: Heterogeneous, Large-Scale, Repetitive Problems

Current domain-independent planners can fail to find solutions to large problems which are composed of smaller, easy problems. Consider the standard Elevator domain, where the task is to transport people between floors with a few elevators (with limited capacity). A small typical instance of the elevator domain, with 3 floors (1F, 2F, 3F) and 1 elevator, is drawn in the left side of Fig. 1. Problems of this scale are easily solved by current domain-independent planners. In the IPC benchmark instances, the number of passengers is at most 7 times the number of elevators in the satisficing track, and much fewer for the optimization track.

Now, consider the much larger instance drawn in the right side of Fig. 1, which might represent what happens at a busy office building when all but one of the elevators are shut down for maintenance. Very large instances of the Elevator problem similar to this can not be solved by current domain-independent planners (as shown in Sec. 4). It is easy to understand why: For a standard, forward-search based planner, the large elevator problem poses a serious challenge because of an extremely high branching factor (whenever the elevator door is open) and a very deep search tree.

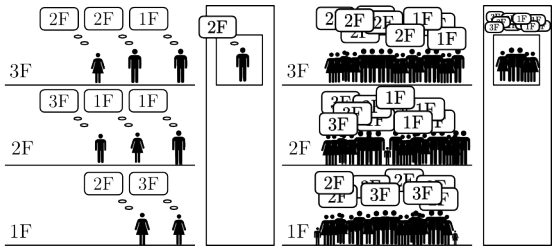

Figure 1: A normal and large Elevator problem instances

On the other hand, a human would have no trouble generating a plan to transport all the passengers to their destinations. When faced with a large problem such as this, a human would not directly search the space of possible action sequences (as current domain-independent planners do). Instead, he/she would notice that on each floor, the passengers waiting on that floor can be divided into two groups, according to their destination. For example, on the 3rd floor, there are passengers who want to go to the 1st floor, and passengers who want to go to the 2nd floor. If there are 3 floors, there can be only 6 kinds of passengers there i.e. $1F \rightarrow 2F$, $1F \rightarrow 3F$, $2F \rightarrow 1F$, $2F \rightarrow 3F$, $3F \rightarrow 1F$, $3F \rightarrow 2F$. If the number of passengers is very large, one natural and fairly efficient solution is a cyclic plan that first transports as many passengers as possible from 1F to 2F, then from 2F to 3F, then 3F to 1F, as many times as needed to transport the $1F \rightarrow 2F$, $2F \rightarrow 3F$, and $3F \rightarrow 1F$ passengers to their destinations. Then we handle the remaining passengers ($1F \rightarrow 3F$, $3F \rightarrow 2F$, $2F \rightarrow 1F$) using a cycles of moves from 1F to 3F, 3F to 2F, and 2F to 1F. More generally, if there are F floors, the passengers on each floor, the passengers on the i-th floor can be partitioned into at most $F-1$ groups, for a total of $F(F-1)$ groups, and a cyclic plan can be similarly constructed. While the cyclic plan can be suboptimal (e.g., if the number of passengers in each group is not equal), this is a reasonably efficient solution.

Other large-scale domains can be approached similarly. If we focus on one type of object (`passenger` in the above example), we can categorize them into groups, by the structural analysis of the initial state (the current floor) and the goal condition (destination floor). This may significantly abstract the basic structure of the domain especially when the number of instances of the object increases faster than the number of the groups does. Note that in our Elevator domain example with 3 floors and 1 elevator, the maximum number of groups is 6, *regardless of the number of passengers*. For a standard planner, increasing the number of passengers makes the problem instance more difficult. However, for an approach that seeks to identify subproblems that can be repeatedly solved using the same method, there is no marginal increase in problem difficulty after a certain point, i.e., the problem difficulty is "saturated".

Furthermore, the interesting things we found is that we can benefit from such categorization even in the standard IPC benchmark problems generated by the official problem generators. Although most subproblems do not form a cycle because they are too different, some *do share* their structure and can form the cycles.

2.1 Cyclic Planning for Homogeneous, Repetitive Problems

To realize these ideas, we adopted a notion of "cyclic planning" and "steady state".

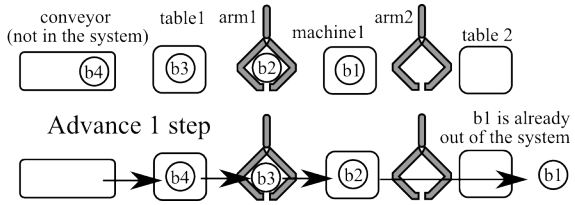

Figure 2: An example of a start/goal state of a cycle, with four products b1,b2,b3 and b4

Cyclic scheduling for robotic cell manufacturing systems, has been studied extensively in the OR literature [Dawande et al., 2005], and the general problem of cyclic scheduling has been considered in the AI literature as well [Draper et al., 1999]. This body of work focused on algorithms for generating effective cyclic schedules for specific domains, and addresses the problem: Given the stages in a robotic assembly system, compute an efficient schedule. Thus, the focus

was on efficient scheduling algorithms for particular assembly scenarios.

Similarly, in a generic planning problem, a *cyclic plan* is a sequence of actions that can be performed repeatedly, and a cyclic planner can gain performance when a planning problem is reduced to a cyclic problem. For example, the most common scenario in CELL-ASSEMBLY based manufacturing plants are orders to make N instances of a particular product. In a cyclic plan, the start and end states of the cycle correspond to a "step forward" in an assembly line, where partial products start at some location/machine, and at the end of the cycle, (1) all of the partial products have advanced forward in the assembly line (2) one completed product exits the line, and (3) assembly of a new, partial product has begun (See Fig. 2). Each start/end state is called a *steady state* for a cyclic plan, which is modeled as a set of partially grounded state variables, e.g., S_i={(at b_{i+2} table), (at b_{i+1} painter), (painted b_{i+1}), (at b_i machine)}. The initial and end state of the cyclic plan includes S_i and S_{i+1}, respectively.

Recently, Asai and Fukunaga developed ACP, a system which automatically formulating an efficient cyclic problem structure from a PDDL problem for assembling a single instance of a product [Asai and Fukunaga, 2014]. ACP takes as input: (1) a PDDL domain model for assembling a single product instance, (2) the type identifier associated with the end product, (3) the number of instances of the product to assemble, N, and (4) an initial state Based only on this input (i.e., without any additional annotation on the PDDL domain), ACP generates a cyclic plan which starts at the initial state, then sets up and executes a cyclic assembly for N instances of the product. If ACP can generate a cyclic plan at all, then (because of its cyclic nature) an arbitrarily large number of instances of the target object can be produced.

While ACP provides one solution for a class of homogeneous, repetitive problems as described above, it is not a complete solution to the problem of automatically generating a cyclic formulation for repeated tasks. There are two significant limitations:

First, ACP can not handle *heterogeneous*, repetitive problems, e.g., a manufacturing order to assemble N instances of one product and M instances of another kind of product. They may or may not share the parts and the jobs, e.g. both kinds of products may require the painting but each requires a different additional treatment such as varnishing and the surfacer treatment.

Second, ACP assumes that all instances of objects are indistinguishable. This limitation can be best explained with a concrete example: In the CELL-ASSEMBLY domain, the task is to complete many products on an assembly line with robot arms (Fig. 2). There are a number of assembly tables and machines that perform specific jobs such as painting a product or tightening a screw. In each assembly table, various kinds of `parts` are attached to a `base`, the core component of the product. For example, a problem requires two kinds of parts, `part-a,part-b`, to be attached to each one of `base0,base1`. The final products look like `base0/part-a/part-b` and `base1/part-a/part-b` . Note that in this example, each `part` is *not* assumed to be associated with any specific

```
(:objects b-0 b-1 - base
          part-a-0 part-a-1 - part
          part-b-1 part-b-1 - part ...)
(:init (part-base part-a-0 b-0)
       (part-base part-a-1 b-1)
       (part-base part-b-0 b-0)
       (part-base part-b-1 b-1) ...)
```

Figure 3: CELL-ASSEMBLY with distinctly labeled parts.

`base`. The two `part-a` 's are treated as if they are supplied as needed and each parts are indistinguishable. While this is acceptable in many cases in the CELL-ASSEMBLY domain, the assumption of indistinguishable instances may not be appropriate in other domains, and even in some CELL-ASSEMBLY scenarios.

Consider a CELL-ASSEMBLY domain where each part is labeled and the problem specifies which specific part instance is attached to which base. Fig. 2.1 describes such a problem i.e. `part-a-0` *must* be attached to `b-0` specifically and so on. Assuming the implementation of ACP system is based on the plan analysis of a unit product, which is one of `b-0` or `b-1` in this case (let it be the former), then the produced cyclic plan contains a parametrized `base` but also a static `part-a-0` object, which leads to inconsistency when we substitute the parametrized `base` with `b-1`. It shows that the information of the `base` objects is not sufficient in order to unroll a cyclic plan: there is a lack of information about the associations between the bases and the parts.

Therefore we need to automatically detect such structure consisting of a core object and the associated objects. Once we find this composite structure, we can treat it as an abstract representation of a unit product. A cyclic plan for the abstract structure can be generated, and when the cyclic plan is "unrolled", individual instances of the parts of the composite object can be mapped to the cyclic plan.

3 Solving Large-Scale, Heterogeneous Repetitive Problems by Decomposition

Given a large-scale problem such as a heterogeneous cell assembly or large-scale Elevator problems described in the previous sections, we can try to solve them by decomposing them into subproblems that can be handled by existing approaches. Our overall approach consists of following 5 steps:

1. Divide the set of objects in a problem by extracting abstract type [2004] information, based on the structural analysis on the initial state. Instances of an abstract type are called abstract components and a PDDL object is allocated to each *slot* defined by abstract type.

2. Extract the initial state and the goal condition that is related to each component. A triple of a component, an initial state and a goal condition is called as abstract task.

3. Compute the compatibility between tasks and categorize the tasks into groups such that each task in a same group share the same plan. *External elements* that are not expressed in a task is considered here.

4. Solve each group of the same tasks as a cyclic planning problem. Each group can be solved separately. We finally get a set of unrolled cyclic plans.
5. Interleave the solutions to the decomposed subproblems. Unrolled plan of each group is interleaved, treated like Abstract Actions in HTN terminology.

The remainder of the paper describes an approach for partitioning of a heterogeneous, repetitive problem into groups of related, easier subproblems, i.e., steps 1-3 above. In particular, we describe a method for generating groups of subproblems that are identical when abstracted (and therefore, only 1 instance from each group needs to be solved).

After the original problem has been partitioned, then Step 4 (solution of subproblems) depends on the partitioning results. If there are numerous instances of each type of subproblem, then a cyclic planner such as ACP can be applied in order to generate an efficient plan for that group of subproblem. On the other hand, groups with a single instance member can be solved using a standard planner.

The final step, combining the subproblem solutions into a single plan for the original problem (Step 5) , is future work. In problems with no (non-replenishable) resource constraints such as the Elevator and CELL-ASSEMBLY domains, a satisficing plan can be obtained by sequentially executing the plans for the subproblems (stiching the end state of each subplan to the initial state of the next subplan may be necessary). In cases where more efficient plans are desired, or if there are resource constraints, combining the subplans is a nontrivial problem which can be at least as difficult as HTN planning. In some resource-constrained cases, our decomposition-based approach may not result in a feasible plan.

3.1 Component Abstraction and Attributes

Our method for identifying subproblems in large-scale, repetitive problems is based on the component abstraction method for macro generation by [Botea, Müller, and Schaeffer, 2004]. In particular, we use only the first sets of components and we extend their notion of an abstract type. This section reviews the method for extracting abstract types described in [2004] as well as discussing the completeness of the approach.

Previous Work: Identifying Abstract Components [2004] The basic idea is to build a *static graph* of the problem and partition it into abstract components by seeding the components with one node from the graph, and then iteratively merging adjacent nodes and "growing" the component. Fig. 4 illustrates a possible static graph and some of its abstract components. Each small circle represents the objects appeared in Fig. 2.1, where "b0" and "pb0" is an abbreviation of `b-0` and `part-b-0` etc. Each color and pattern of a node implies its *type*: different colors suggest different types.

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 graph may be unconnected.

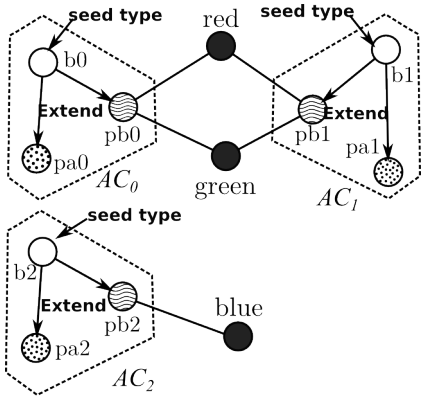

Figure 4: Example of a static graph and components

Each component is a subgraph of the static graph. The decomposition into components proceeds as follows:

1. First, a *Seed Type* is selected (e.g., randomly) In the figure, `base` ○ is selected as a seed type.
2. Next, all objects of the seed type in the static graph are collected, and an abstract component is created for each selected object. In the figure, the seed objects are `b0,b1,b2`, and their corresponding abstract components are `(b0),(b1),(b2)`.
3. A *fringe* node is a node that is currently not included in any components and adjacent to some node in a component. If no such fringe node exists, then it either restart the search with the other randomly-selected seed type and the rest of the graph, or terminates if no other seed types remain.
4. Select a set of fringe nodes simultaneously, choosing a type and then selecting all fringe nodes of that type. For example, if we choose a wave-patterned node `pb0` ◍ for the white node `b0` ○, then we simultaneously choose `pb1` ◍ for `b1` ○ and `pb2` ◍ for `b2` ○. The selection order is not specified.
5. Merge the selected nodes into the component that they are adjacent to, e.g., if we choose `pb0` ◍ first, then the components are updated from `(b0),(b1),(b2)` to `(b0 pb0),(b1 pb1),(b2 pb2)`.
6. At this point, check if the resulting components share any part of the structure, and if so, discard all the fringe nodes newly added in step 5. For example, extending the top-left structure from `pb0` ◍ and adding `red` ● simultaneously causes the top-right structure to include `red`, resulting in sharing `red`. Since the objects merged in the step 5 are `red`, `green` and `blue`, they are all excluded.

Towards a more Systematic Search for the Best Abstraction The original abstract type detection by Botea et al is a randomized, greedy procedure. The Seed Type is randomly selected in step 1 and step 3. In addition, in step 4, the type of fringe nodes is selected arbitrarily (the selection order was not specified in [Botea, Müller, and Schaeffer, 2004].) While this kind of randomized, greedy algorithm was compatible

with their goal of quickly extracting macro-operators, this is not appropriate for our purpose (solving large problems by decomposition), due to two reasons.

First, depending on the choices made regarding Seed Type selection and fringe node type selection, we may fail to find a component abstraction that includes any nodes other than the seed node. In the case of macro abstraction, such a failure is not necessarily fatal because the macro system is basically a speedup mechanism, and even if a macro is not found, the base-level planner may still be able to solve the problem. On the other hand, our objective is to solve problems that are completely beyond the reach of standard planners (with hundreds of thousands of objects), so failure to find an appropriate abstract component is equivalent to failure to solve the problem at all.

Second, there may be multiple possible component abstractions for any static graph, and for our purpose (solving very large problems by decomposition), it is not sufficient to find *any* component abstraction. For example, consider the integration of component abstraction into the ACP cyclic planner. ACP extracts *lock* and *owner* predicates associated with each object. A "good" component abstraction in this context is one which results in the extraction of a large number of *locks*, which, in turn, results in efficient plans with good parallel resource usage. Since *locks* are extracted for each object and they are aggregated for each component which contains the object, large components tends to have a large number of *locks* in general.

It is straightforward to modify the abstract type detection algorithm to systematically enumerate and consider all possible component abstractions that can be extracted from a static graph. However, we don't currently perform a full enumeration. We run the abstract component detection algorithm from a fresh initial state for all possible seed types. We only use the *first* set of components that has been extended from the *first* seed type given in the initial input. Thus, step 3 is modified so that "continuing the extraction in the remainder of the graph" is omitted. While this is more systematic than the original algorithm by Botea et al, the choice of fringe-node type is still arbitrary (simple queue with no priority). A more systematic approach to searching for good abstract components (including reducing the search effort by pruning the space of candidate component abstractgions) is future work.

Attributes Conceptually, an abstract component represents an inseparable groups of objects such as a name, arms and legs of a human. The fact that an arm belongs to a person is never removed or added (or it means the transplantation or loss of the arm). Also, an arm does not belong to more than one person (again except a few cases).

In contrast, some nodes in the static graph represent *Attributes* that belong to many groups of objects, e.g. hair color, ethnicity, and gender are attributes that are shared among people.

We extend the abstract type detection algorithm of [2004] above to identify such attributes. In Step 6 above, all nodes that prevented the extension are identified as *attributes*. In the example, `red`,`blue`,`green` are attributes. Attributes are used in order to constrain the search for plan-compatible abstract tasks, described below.

3.2 Abstract Task

We define an *abstract task* as a triple consisting of (1) an abstract component 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* in the problem description. Fluent facts are the facts which are not static. For each abstract component, we collect all fluent facts in the initial and goal states which contain one of the objects in AC in its parameters. For example, the initial state in the PDDL model in Fig. 2.1 may include a fact `(not-painted b0)`. It can be removed by some actions like `paint`, so it is fluent. Since `(not-painted b0)` has `b0` in its argument, it is considered as one of the initial states of the top-left component (AC_0) in Fig. 4. Similarly, facts like `(at b0 table1)`, `(at pa0 tray-a)` are the possible candidates for the initial states of AC_0. Likewise, the goal condition of AC_0 may be `(at b0 exit)`,`(is-painted pa0 red)`,`(assembled b0 pa0)` etc.

After all the abstract tasks are identified, we check the compatibility between the tasks. The compatibility is checked via replacing the parameters in a plan with the objects in a component. This finally allows the objects to be correctly categorized, as the people on each floor in Elevator domain were divided into groups.

Plan-wise Compatibility of Tasks In order to define the plan-wise compatibility of tasks, we first define a notion of *component plan* and its compatibility.

A component plan of an abstract component X is a result plan of *component problem* of X. Let the original planning problem is $\Pi = \langle \mathcal{D}, O, I, G \rangle$ where $\mathcal{D}$ is a domain, O is the set of objects and I, G is the initial/goal condition. A component problem is a planning problem $\Pi_X = \langle \mathcal{D}, O_X, I_X, G_X \rangle$ where:

$$
\begin{aligned}
O_X =& X \cup \{O \setminus X \ni o \mid \forall Y \neq X; o \notin Y\} \\
I_X =& \{I \ni f \mid \mathit{params}(f) \cap (O \setminus O_X) = \emptyset\} \\
G_X =& \mathit{goal}(X)
\end{aligned}
$$

The definition of O_X specifies that it removes any objects that belong to another component Y. Note that the objects not included in any component remain as it is. I_X specifies that an initial condition is removed when it has a removed object in its parameters list. G_X is same as the goal conditions of the abstract task of X.

We solve a component problem Π_X with a domain-independent planner such as Fast Downward. However, in some domains no solutions exist in Π_X due to the removed objects and initial conditions. In such cases, we restore O and I and re-run the planner on $\langle \mathcal{D}, O, I, G_X \rangle$. This is called an *Object Restoration*. The computation of a component plan is instantaneous in most cases, but large instances tend to requires more time and memory. As shown in the experiments, we found that when a large number of unused objects

is restored in O, the PDDL $\rightarrow$ SAS converter in Fast Downward can become the bottleneck.

Solving component problem yields an *component plan*. The component plans P_X, P_Y are compatible if P_X can be used as a plan of Y by replacing the parameters i.e. if we replace all references to the objects in X in P_X with the corresponding objects in Y, then the modified (mapped) plan P'_X is a valid plan of Π_Y.

Finally, we define two tasks are *plan-wise compatible* when the following conditions hold:

1. The components of the two tasks are of the same abstract type.
2. The two tasks shares the same set of attributes.
3. The graph structure designated by their initial/goal condition are isomorphic, just like abstract-type, e.g. if the task of AC_0 in Fig. 4 has an initial state `(at b0 table1)`, then a compatible task of AC_1 should also contain `(at b1 table1)` in its initial state.
4. One of the *component plans* of the component problem of each task are compatible.

We categorize the extracted abstract tasks according to the plan-wise compatibility. However, solving the component problems involves running a domain-independent planner, so computing component plans for every abstract task should be avoided if possible. Thus, we use attributes (condition 2 above) in order to filter the candidate pairs before checking whether their component plans are compatible.

Categorizing a group of N elements based on a equality function requires $O(N^2)$ comparisons between the elements in a naive implementation. However, in the IPC problems, most tasks are not compatible, which means they end up in many groups of only a single task in it. Clearly, a group of one task requires no further categorization. A similar observatoin can be made regarding condition 3.

The use of attribute-based comparison greatly reduces the number of pairs whose component plans need to be compared. Consider three similar factory assembly tasks, t_1, t_2, t_3, that require painting of an object with different colors (blue, red, green). Also, assume that the available colors in each painting machine is limited e.g. machine m_1 supports red only and m_2 supports green and blue. Now the plans for t_1 and t_2 are incompatible because they use the different machines. We can detect these incompatibilities without spending all the effort of running a planner, getting a mapped plan and validate it.

While we show experimentally that pre-categorization according to attributes is was highly effective (Sec. 4.2), this method is subject to false-negatives, and can result in the compatible pairs being discarded. Continuing the previous example, plans for t_2 and t_3 is likely to be compatible because their plans use the same machine. However, once we use the condition 2, it divides t_2 and t_3 into the different groups because they use the different colors.

The number of calls to the underlying planner can also be optimized. Since the compatibility is transitive (given the condition 2 and 3. Proof omitted), we can check the compatibility between Y and Z by instead just checking the compatibility between X and Z. Then we can reuse P_X because the compatibility check requires only the plan P_X (and does not require P_Z).

4 Experimental Results

We evaluate abstract-task based problem decomposition on the CELL-ASSEMBLY domain as well as large instances of several IPC domains.

4.1 Categorization of CELL-ASSEMBLY Problems

We evaluated our decomposition method on both homogeneous and heterogeneous CELL-ASSEMBLY-EACHPARTS. Both types of problems are defined on the same manufacturing plant with the same tables and machines. However, the latter processes two completely different kinds of products at the same time. The results are shown in Fig. 5.

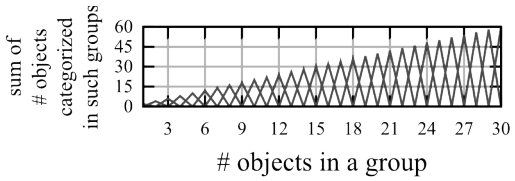

Figure 5: The categorization result of CELL-ASSEMBLY-EACHPARTS *2a2b-mixed* problems with seed `base`. Each problem contains n 2a-tasks and the same n 2b-tasks, where $1 \leq n \leq 30$. Each line represents one problem. The x-axis represents the number of objects in each categorized group, while the y-axis represents the total number of objects in the groups of x objects. Therefore $(x, y) = (15, 30)$ means there are 2 groups of 15 components. For each problem, the tasks are divided into 2 groups of n tasks, where each group represents *2a* and *2b*. This shows that our algorithm correctly categorized two groups of tasks in each problem, despite the combination of the mixed orders and the variation within each group (Sec. 2.1).

On the homogeneous CELL-ASSEMBLY-EACHPARTS problems, our algorithm correctly identifies all component tasks and labels them as plan-wise compatible. On the heterogeneous problems, it identifies that there are two kinds of tasks and the objects are compatible within each group. (When we say "heterogeneous $x + y$ problem", it contains x instances of one product and y instances of another product.) Decompositions based on the seed types other than `base` was also successful. The seed was `part`, and the detailed analysis suggested that the same abstract type was detected from the different seed types. Given this fact, we consider the exhaustive attempts on all seed types are necessary.

4.2 Categorization results for IPC Domains

We also evaluate our decomposition method on several IPC domains including Satellite (minimally modified, see below), woodworking, openstacks, elevators, barman, and rover. Satellite-typed is a typed variant of IPC2006 satellite. While the original domain is already "typed" by the use of 1-argument type predicates, we simply converted the type

predicates to standard, explicit PDDL type annotations. This conversion has no effect on the structure of the domain or instances. Although we performed this modification manually, an automatic procedure such as TIM[Fox and Long, 1998] could have been used instead. Very large problem instances for the IPC domains were generated using either the scripts that are found in IPC 2011 result archive [1] (for Woodworking, Barman and ROVER), and our own generator for the rest.

First, in order to understand the limitations of current State-of-the-Art planners, we run the current version of Fast Downward[2] using the "seq-sat-lama-2011" satisficing configuration, which emulates the IPC2011 LAMA planner. The maximum search time is limited to 6 hours, and memory limit of $15[\mathrm{GB}]$ on an Intel Xeon E5410@2.33GHz.

The table below shows the largest problem instances that was solved. Stars ($*$) mean it uses the generators and the problems used in IPC. (In this table, we also summarized the seed types which are used in the later categorization.)

Domains	limit	seed type
cell-assembly	(single product, 14 bases)	base,part
-eachparts	(mixed products, 6+6 bases)	
satellite-typed	(17 satellites,39 instruments,	direction
	13 modes,310 directions)	
IPC domains		
woodworking*	p86 (227 parts and x1.2 wood)	part
openstacks	(70 orders and products)	order, product
elevators	(4 slow&fast elevators, 40 floors,	passenger
	270 passengers, 10 floors/area)	
barman-sat11*	(4 ingredients, 93 shots and cocktails)	shot,cocktail
rover*	N/A (IPC problems were solved up to p40)	objective

Table 1: Domains used in the evaluation.

Assuming that once a cyclic plan is generated, the marginal cost of processing an additional product within a group is *zero*, we can summarize the effect of categorization by counting the number of groups and comparing it with the number of components. Fig. 6 shows the result.

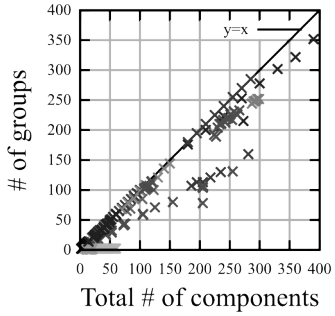

Figure 6: The number of components vs the number of categorized groups. Results of all domains categorized using all seed types are shown at once. Points from the different configuration have the different color(density).

[1] svn://svn@pleiades.plg.inf.uc3m.es/ipc2011/data

[2] http://www.fast-downward.org

The points on the line $x = y$ in Fig. 6 shows that we cannot get a meaningful categorization in some configurations. The reason is twofold: Firstly, the choice of *seed* was inappropriate. Indeed, in `barman` domain, categorization based on `cocktail` did not result in components larger than the initial node, while that of `shot` did. This is due to the greedy nature of the abstract type detection algorithm, as described in [Botea, Müller, and Schaeffer, 2004]. The same thing applies to the results of the other seed types (though not shown here), such as the decomposition of CELL-ASSEMBLY-EACHPARTS based on `arm`, `table`, `job` and so on. This indicates that restarting the search with all seeds as described in Section 3.1 is required to ensure that a meaningful categorization is found if one exists. Secondly, in OPENSTACKS, both seeds `order` and `product` failed to get a meaningful information. In this case, the reason is attributed to the domain's characteristics itself i.e. it is possible that the domain has no inherent meaningful categorization.

Evaluation of the Optimization Methods Here we show the effect of pre-categorization described in Sec. 3.2. Though we are not able to include detailed results here, Fig. 7 (left) shows the distribution of pre-categorization (based on the conditions 1 to 3, Sec. 3.2). It shows that the number of comparisons that are actually performed is far less than those required by the naive method because most categorizations are already done i.e. most tasks are already categorized into groups of a few elements (1 or 2 elements).

We also compare the actual number of calls to the underlying planner with a naive implementation. In a naive implementation, N components require N calls to the planner. Fig. 7 shows the results in all domain shown in Table 4.2.

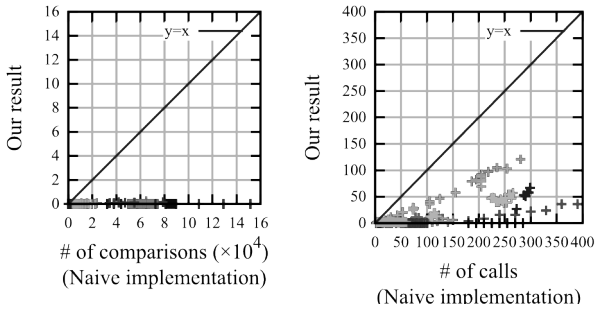

Figure 7: Cumulative evaluation results among all problems. Points from the different configuration have the different color and density. (Left) Our pre-categorization method significantly reduced the number of comparisons necessary to categorize the tasks. (Right) Our memoization strategy significantly reduces the number of calls to the underlying planner by a factor of ≈ 2.

Total Elapsed Time of the Categorization Elapsed time is a key factor when our method is considered as a preprocessing step for planning. We show the results in Fig. 8 (left). Unfortunately, some problems takes very long time to compute the categorization. The problem is caused by the object restoration (Sec. 3.2). Fig. 8 (right) also shows the num-

ber of object restorations occured during the decomposition, which clearly indicates that in the most time consuming ($t \approx 10^5$) decompositions, most tasks are evaluated by a component problem with object restoration ($\langle \mathcal{D}, O, I, G_X \rangle$), not by $\langle \mathcal{D}, O_X, I_X, G_X \rangle$.

In the same figure (right), we show that the current bottleneck in solving a restored problem is mainly the PDDL $\rightarrow$ SAS converter in Fast Downward. Addressing this issue remains future work.

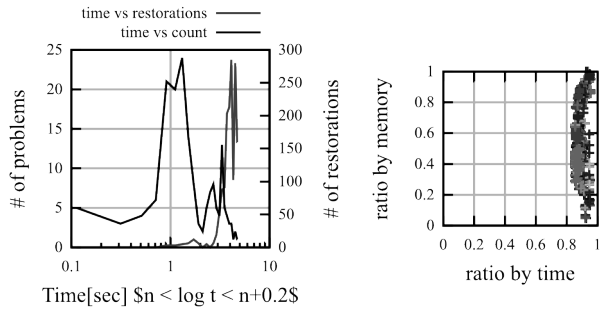

Figure 8: (Left) Histogram of the total elapsed time[sec], averaged in $n \leq \log t \leq n + 0.2$ for each n. (Right) x-axis is a ratio of translate/(translate + preprocess + search) measured by time, while y-axis is measured by memory. We show the results from all the domains, regardless of whether the object restoration has occured. It clearly shows that the search in a component problem is easy because of the limited number of conditions in G_X, and the computation is usually dominated by the translator.

5 Related Works

The overall approach we are pursuing, which is to decompose a large problem into relatively easy subproblems is inspired by previous work on problem decomposition [Yang, Bai, and Qiu, 1994] and HTN planning [Erol, Hendler, and Nau, 1994].

Our approach is related to macro abstraction systems such as Macro-FF [Botea et al., 2005], which automatically identifies reusable plan fragments, and in fact, the component abstraction framework we extended was originally used by Botea et al to identify macros [Botea, Müller, and Schaeffer, 2004]. Macro systems strive to provide a very general abstraction mechanism, but are typically limited to relatively short macros (e.g., 2-step macros in Macro-FF). Our approach, on the other hand, focuses on identifying 'very long macros" (10-30 steps per cycle) corresponding to specific abstract tasks.

6 Conclusions

This paper presents preliminary work on a decomposition-based approach for solving large scale planning problems with a repetitive structure in domains such as factory assembly. We presented an overview of our decomposition-based approach, and described a novel method for automatically detecting such a repeated structure, based on categorization of objects in a problem according to abstract task compatibility. We showed experimentally that on very large problem instances, our method is able to able to successfully decompose the problems into tasks that satisfy a particular compatibility criteria (plan-wise compatibility). We showed that plan-wise compatibility can be found not only in large instances of the factory assembly domain that is the primaryh motivation for this line of work, but also in large instances of IPC2011 domains.

In the overall approach to solving large-scale, heterogeneous repetitive problems outlined in Section 3, this paper addresses Steps 1 and 3. Step 4 (cyclic planning) has already been addressed in [Asai and Fukunaga, 2014]. The next step is to address the remaing steps (2, 5), which would result in a system that can fully automatically approach the problem of solving large-scale repetitive problems.

There are several directions for improving the decomposition strategy proposed in this paper. First, as discussed in Section 3.1, the abstract type detection algorithm should be made more systematic, and a focused search for larger components should be implmented. In addtion, a smarter object restoration strategy which minimize the number of objects to be restored (Section 3.2) because currently all objects and initial configurations are restored again in $\langle \mathcal{D}, O, I, G_X \rangle$, causing the underlying planner to be confused by the unnessesarily large number of objects.

References

Asai, M., and Fukunaga, A. 2014. Fully automated cyclic planning for large-scale manufacturing domains. In *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 ICAPS*, 181–190.

Bylander, T. 1994. The computational complexity of propositional strips planning. *Artificial Intelligence* 69(1):165–204.

Dawande, M.; Geismar, H. N.; Sethi, S. P.; and Sriskandarajah, C. 2005. Sequencing and scheduling in robotic cells: Recent developments. *Journal of Scheduling* 8(5):387–426.

Draper, D.; Jonsson, A.; Clements, D.; and Joslin, D. 1999. Cyclic scheduling. In *Proc. IJCAI*.

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. *Journal of Artificial Intelligence Research*.

Ochi, K.; Fukunaga, A.; Kondo, C.; Maeda, M.; Hasegawa, F.; and Kawano, Y. 2013. A steady-state model for automated sequence generation in a robotic assembly system. *SPARK 2013*.

Yang, Q.; Bai, S.; and Qiu, G. 1994. A framework for automatic problem decomposition in planning. In *AIPS*, 347–352.