Finding Solution Preserving Linearizations For Partially Ordered Hierarchical Planning Problems

Ying Xian Wu[1], Songtuan Lin[1], Gregor Behnke[2], and Pascal Bercher[1]

School of Computing, The Australian National University, Canberra, Australia[1]
ILLC, University of Amsterdam, Amsterdam, The Netherlands[2]
{yingxian.wu, songtuan.lin, pascal.bercher}@anu.edu.au, g.behnke@uva.ni

Abstract. Solving partially ordered hierarchical planning problems is more computationally expensive compared to solving totally ordered ones. Therefore, automatically transforming partially ordered problem domains into totally ordered ones, such that the totally ordered problem still retains at least one solution, would be a desired capability as it would reduce complexity and thus make it easier for planning systems to solve the problem. This is a complex endeavour, because even creating *all* possible linearizations of all methods in the original domain does not guarantee that solutions are preserved. It also allows the planner to use algorithms and heuristics specialised for the totally ordered case to solve the transformed problem. In this paper, we propose an algorithm for converting partially ordered problems into totally ordered ones and give criterion for when this is possible. We test our techniques on the partially-ordered track of the bench-mark set of the IPC 2020 and solve both the linearized and the original partially-ordered problems using state-of-the-art planning systems. We find that in the majority of problems across a variety of domains, the linearized problem remains solvable, and can always be solved faster than the without our proposed pre-processing technique.

Keywords: Partially ordered HTN planning · Hierarchical planning · Totally ordered HTN planning

1 Introduction

Hierarchical Task Network (HTN) planning [2] [6] is a hierarchical approach to planning. Tasks in HTN planning are either primitive, corresponding to an action that can be taken, or compound. HTN problems have a set of methods that specify how one might achieve a given compound task, by decomposing it into a set of sub-tasks. A compound task may even decompose into itself, either directly via a method, or indirectly via a sequence of method applications. If decomposition leads to a sequence of primitive tasks executable from the initial state, then this sequence of actions is a solution to the problem, also known as a "plan".

In totally ordered HTN planning, or TOHTN planning, methods specify a total order on the sub-tasks. In partially ordered HTN planning, or POHTN planning, methods might only specify a partial order on the sub-tasks.

Certain kinds of problems might be naturally more suited to being modelled as a partially ordered problem, for example, the actions “deliver package 1 to city A” and “deliver package 2 to city B” – these are essentially unrelated goals in a real transport scenario, and so modelling the problem to require that one task be completed before the other would be unnecessarily limiting the possible solution space. In some cases, such over-specification may remove all valid solutions. Thus many problems might be *modelled* as POHTN problems. However, for *solving* the problem, it is desirable to have additional constraints that reduces the search space, while still preserving at least some of the actual solutions.

This paper specifically presents a method to transform a POHTN problem to a TOHTN problem in order to reduce the search space. Converting the problem to a TOHTN problem allows us to exploit the fact that as proven in Erol et al. [4], TOHTN planning as a class of problems has lower computational complexity, resulting in lower worst-case solving time. Thus, transforming a POHTN problem to a TOHTN problem could allow us to solve the problem more quickly and deploy specialised algorithms and heuristics.

The drawback to this approach is that, due to the greater expressivity of POHTN planning, there may exist POHTN problems that cannot be solved when converted to a TOHTN problem. For example, Erol et al. [4] proved that HTN planning is expressive enough to model undecidable problems, such as the language intersection problem of two context-free languages. Fortunately, not every POHTN problem is guaranteed to be undecidable, and so could still be transformed while preserving at least one solution.

In this paper we present and investigate an algorithm for converting POHTN problems to TOHTN problems. We prove that when certain criteria are met, it guarantees that at least one solution will be preserved. Also, we obtain a new class of decidable problems, namely those that satisfy the above mentioned criterion. Finally, we show that, even when these criteria are not met, very few problems are rendered unsolvable by the transformation, and that it greatly reduces solving time for problems, with gains being bigger for more difficult problems.

2 Hierarchical Planning

We first introduce classical planning, as hierarchical planning can be considered an extension of classical planning.

2.1 Planning Problem Formalisation

Classical planning problems are defined over a **domain** $\mathbf{D} = (F, A)$, where F is a finite set of facts, or propositional state variables. A is a finite set of actions. For all $a \in A$, $a \in 2^F \times 2^F \times 2^F$, which represents the preconditions and add and delete effects of an action. The preconditions, add, and delete effects, of an action a are referred to as $pre(a)$, $add(a)$, and $del(a)$ respectively. An action a is executable in a state s if its precondition pre holds in s, i.e. $pre \subseteq s$. If executable in s, its result is the successor state $s' = (s \backslash del) \cup add$, i.e., variables in $del(a)$ get removed and variables in $add(a)$ get added.

A **classical planning problem** is defined as $\mathbf{P} = (D, S_I, S_G)$, where D is the domain of the problem, $S_I \in 2^F$ is the initial state and $S_G \in F$ is the goal description. Given a state $s \in 2^F$, a fact is known for certain to be either true or false, i.e. information about the world state is complete at all times (this is called the closed world assumption).

A **solution** $\bar{a}$ to a problem is any action sequence that is executable in the initial state S_I, and leads to a state s'' that satisfies all goals, i.e., $s'' \supseteq S_G$. Any such state s'' is called a goal state.

There are many formalisations for hierarchical planning, also known as **HTN planning**. The following one borrows heavily from Bercher et al. [2] and Geier and Bercher [5]. HTN planning has two variants, Partially Ordered Hierarchical Task Network Planning, also known as POHTN planning, and Totally Ordered Hierarchical Planning, also called TOHTN planning,

A POHTN problem $\mathbf{P} = (D, S_I, T_I)$ is defined over some domain D, has an initial state $S_I \in 2^F$, and has a initial compound task T_I. The closed world assumption also holds for HTN planning.

The domain $\mathbf{D} = (F, T_P, T_C, \delta, M)$, where F is the finite set of facts or state variables, T_P is the finite set of all possible primitive task names, T_C is the finite set of all possible compound task names, and δ is a mapping from primitive task name to action. Actions in POHTN domain also have preconditions, adds, and delete effects. M is the finite set of decomposition methods. Each one maps a compound task name to a task network. If $m \in M$, then $m = (c, tn)$, where $c \in T_C$.

A task network $\mathbf{tn} = (T, \prec, \alpha)$ consists of T, which is a finite set of task identifiers (ids); $\prec$, which is a partial order over T; and α, which maps task ids $\in$ T to task names in T_C and T_P.

A method m decomposes a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ into a new task network tn_2 by replacing t, if and only if $t \in T_1$, $\alpha_1(t) = c$, and $\exists tn' = (T', \prec' \alpha')$ with $tn' \cong tn_m$ and $T' \cap T = \emptyset$ and

$$tn_2 := ((T_1 \setminus \{t\}) \cup T', \prec_1 \cup \prec' \cup \prec_X, \alpha_1 \cup \alpha') \tag{1}$$

$$\prec_X := \{(t_1, t_2) \in T_1 \times T' \mid (t_1, t) \in \prec_1\} \cup \tag{2}$$

$$\{(t_1, t_2) \in T' \times T_1 \mid (t, t_2) \in \prec_1\} \tag{3}$$

In other words, the decomposition of a compound task results in it being removed from the task network and replaced by a copy of the method's task network. The ordering constraints on the removed task are inherited by its replacement tasks, as defined by $\prec_X$.

A solution to an HTN problem is a task network $tn = (T, \prec, \alpha)$ if and only if tn can be reached via decomposing tn_I, all tasks are primitive, $(\forall t \in T : \alpha(t) \in T_P)$, and there exists a sequence $\langle t_1, t_2 ... t_n \rangle$ of the task ids in T that agrees with $\prec$ such that the application of that sequence $\langle \alpha(t_1), \alpha(t_2) ... \alpha(t_n) \rangle$ in S_I is executable.

In other words, the goal of hierarchical planning is to find an decomposition of the task, then any executable refinement of the resulting decomposition. Whereas in classical planning, one only finds any executable sequence of actions to achieve a goal state, so HTN planning poses additional restrictions on which action sequences may be considered.

Totally Ordered Hierarchical Planning, also called TOHTN planning, is the same as partially ordered planning in all respects except the kind of task networks it allows. For both planning formalisms, a method m maps a task t to a task network $\mathbf{tn} = (T, \prec, \alpha)$. TO planning domains require that $\prec$ must specify a total order between task ids in T. This leads to a difference in expressiveness and decide-ability of TOHTN vs POHTN planning. POHTN planning is more expressive in general (both in terms of plan existence and in terms of computational complexity). As per Höller et al. [9], if regarded from the standpoint of formal grammars, TO planning is exactly as expressive as context free languages, whereas PO planning is strictly more expressive than context-free languages, and strictly less expressive than context-sensitive languages. In terms of complexity classes, Erol et al. [4] proved that POHTN planning is semi-decidable, whereas Alford et al. [1] proved that, assuming arbitrary recursion, TOHTN planning is 2-EXPTIME-complete with variables, and EXPTIME-complete without.

3 Linearization Method

Though this will not always be possible, we want to impose a total order on the task networks, such that for a method $m = (c, (T, \prec, \alpha))$, the decomposition of t will result in an executable sequence. To attempt this, one could analyse what preconditions the execution of sub-tasks in T might need, and what effects they might have. For example, if a sub-task t deletes a fact in the precondition for sub-task t', one could order t' before t to avoid invalidating the precondition for t'. So we need a way of estimating preconditions and effects for compound tasks. It was proven by Olz et al. [15] that inferring all preconditions and effects of a compound task is as difficult as solving the problem. Therefore, a polynomial-time algorithm to infer the exact set of preconditions and effects of a compound task does not exist. However, using an approximated set, one can analyse the proposed orderings and resolve any conflicts, e.g. cycles. If conflict-resolution is necessary, we refer to this as needing ***cycle-breaking***.

The algorithm provided in this paper approximates possible preconditions and effects for all tasks so that this information can be used to linearize methods. For a given task t, we call these approximate preconditions and effects as $pre^*(t), del^*(t), add^*(t)$. If t is an action, $pre^*(t), del^*(t), add^*(t)$ is the same as $pre(t), add(t), del(t)$. If t is a compound task, $pre^*(t)$, $add^*(t), del^*(t)$, are the union of the respective preconditions, add, and delete effects of all actions that t could decompose to, as shown in Figure 1. This means that for a task t $pre^*(t), del^*(t), add^*(t)$ can contain contradictory effects, e.g. t adds *and* deletes a fact. These are essentially the mentioned literals defined by de Silva et al. [17].

Figure 2 shows how the additional orderings for a method are determined. We then attempt to integrate these orderings into the method $m = (t, (T, \prec, \alpha))$. If there exists a cycle after integrating the new orderings into $\prec$, we remove a random one of the new orderings in that cycle, as shown in Figure 3, until that cycle no longer exists. We repeat this for all methods in order to linearize them.

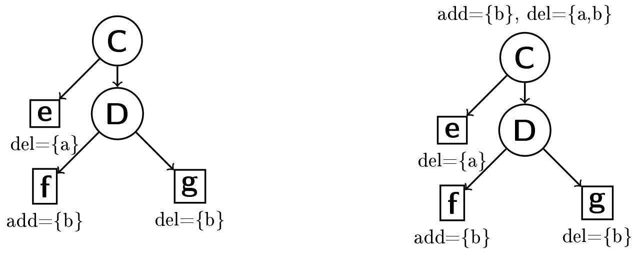


Fig. 1: Inferring preconditions and effects for compound tasks

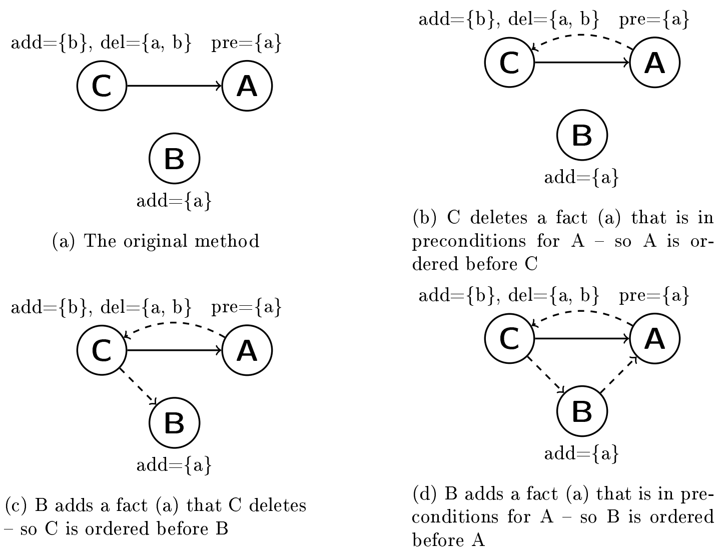
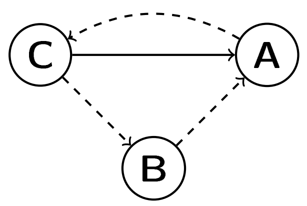
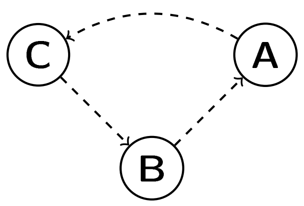

add={b}, del={a, b}
pre={a}
C
A
B
add={a}
(a) The original method
add={b}, del={a, b}
pre={a}
C
A
B
add={a}
(b) C deletes a fact (a) that is in preconditions for A – so A is ordered before C
add={b}, del={a, b}
pre={a}
C
A
B
add={a}
(c) B adds a fact (a) that C deletes – so C is ordered before B
add={b}, del={a, b}
pre={a}
C
A
B
add={a}
(d) B adds a fact (a) that is in preconditions for A – so B is ordered before A

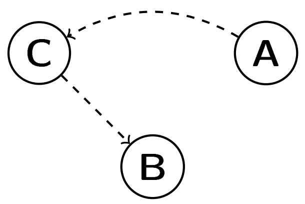
Fig. 2: Adding possible orderings to methods



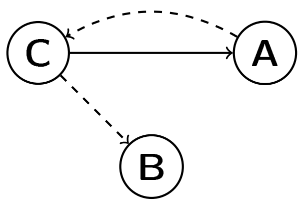
(a) Perform depth-first search on the modified method



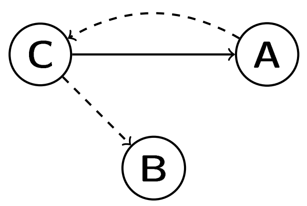
(b) Identify cycle (path along which a node is reachable from one of their ancestors)



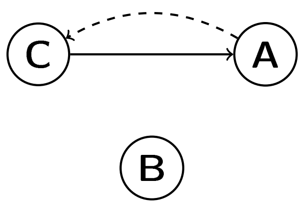
(c) Pick an edge not originally in the method (i.e. a dashed line edge) and delete it.



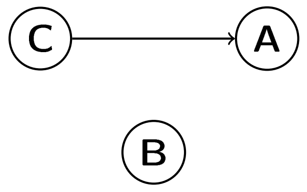
(d) Repeat as necessary until depth-first search cannot find any path back to a previously visited node



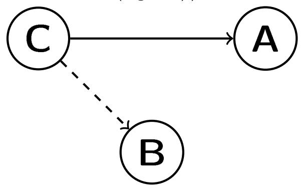
(e) Perform depth-first search on the modified method (again)



(f) Identify cycle (path along which a node is reachable from one of their ancestors (again))



(g) Pick an edge not originally in the method (i.e. a dashed line edge) and delete it (again).



(h) No more cycles, so perform a topological sort to produce a total ordering that satisfies the above constraints

Fig. 3: Cycle breaking to make method linearized

3.1 Code

$$\forall t \in T_P : pre^*(t) = pre(t) \wedge add^*(t) = add(t) \wedge del^*(t) = del(t)$$
$$\forall t \in T_C : pre^*(t) = \{f \mid \exists m \in M : m = (t, (T, \prec, \alpha)) \forall t' \in T . \forall f \in pre^*(t')\}$$
$$\forall t \in T_C : add^*(t) = \{f \mid \exists m \in M : m = (t, (T, \prec, \alpha)) \forall t' \in T . \forall f \in add^*(t')\}$$
$$\forall t \in T_C : del^*(t) = \{f \mid \exists m \in M : m = (t, (T, \prec, \alpha)) \forall t' \in T . \forall f \in del^*(t')\}$$

```
   Data: $(F, T_P, T_C, \delta, M)$
   Result: $(F, T_P, T_C, \delta, M)$
1  for $m = (t, (T_m, \prec, \alpha)) \in M$ do
       /* An edge $(t, t')$ in $G$ means $t$ is ordered before $t'$      */
2      $G \leftarrow \prec$
3      for $a \in F$ do
4          for $t \in T_m$ do
5              for $t' \in T_m$ do
6                  if $a \in add^*(t)$ and $a \in pre^*(t')$, add $(t, t')$ to $G$
7                  if $a \in add^*(t)$ and $a \in del^*(t')$, add $(t', t)$ to $G$
8                  if $a \in del^*(t)$ and $a \in pre^*(t')$, add $(t', t)$ to $G$
9                  if $a \in del^*(t)$ and $a \in add^*(t')$, add $(t, t')$ to $G$
10     while $G$ has cycles in it do
11         Delete a random ordering in $G$ that is not in $\prec$
12     $\prec' =$ Any linearization of $G$
13     $m' = (tasks(m), \prec', \alpha(t))$
14     $M' = M' \cup \{m'\}$
15 return $D' = (F, T_P, T_C, \delta, M')$
```

Algorithm 1: Calculation of linearized methods

3.2 Runtime

Theorem 1. *Given a problem $P = (F, T_P, T_C, \delta, M)$, Algorithm 1 takes at most quadratic time, $\mathcal{O}(|M| * |F| * |T|^2)$.*

Proof. To calculate pre^*, add^*, del^* for each task t, we can perform breadth-first search on the task decomposition sub-tree (where tasks are nodes, and edges indicate possible decomposition by methods) rooted at t. The size of a sub-tree has an upper limit of $T_C + T_P$ nodes. For each primitive task in the sub-tree, we can iterate over each fact to update pre^*, add^*, del^* for the root t. Thus calculating pre^*, add^*, del^* for a single compound task has an upper limit of $(3 * |F| * |T_p|) + |T_C|$. This inference occurs for each compound task, so inference for all compound tasks takes $((3 * |F| * |T_p|) + |T_C|) * |T_C|$ time at most, or $\mathcal{O}(|F| * |T|^2)$, where $|T|$ refers to $|T_P| + |T_C|$.

Lines 5 to 12 of Algorithm 1 builds a graph to represent each method. This graph's nodes are tasks of the task network produced, and the edges represent orderings between tasks. The code here iterates over every method, which iterates over every fact, which iterates over every sub-task in that method, which iterates over every other sub-task in that method. Leading to $(M*F*(t_m)^2)$, where t_m is the average number of sub-tasks per method. So it's at most $\mathcal{O}(|M|*|F|*|T|^2)$, since t_m has an upper bound of T, where $T = T_p + T_C$.

Lines 12-13 of Algorithm 1 can be done by Depth-First Search (DFS). If a "back-edge", defined as the edge that leads back to an already visited node (indicating a loop), then this edge is deleted, providing it was not part of the original domain. If it was part of the original domain, then assuming that the back edge was from node A to B, then a random edge is selected along any path from B to A (i.e. from the other part of the loop) and deleted instead. DFS is known to be in $\mathcal{O}(|V|)$, and finding a path back can be achieved using Dijkstra's algorithm, which is known to be in $\mathcal{O}(|V|^2)$.

Temporal complexity of removing cycles is therefore $(|M|*((t_m+t_m^2)*c)) = M*{t_m}^2*c$, where t_m is the number of sub-tasks produced by a method, and c is the number of cycles. This is approximately $\mathcal{O}(M*T^2*c)$, since t_m is upper bounded by T, where $T = T_p + T_C$.

Line 14 can be done via topological sort of the graph (which is known to be in $\mathcal{O}(V+E)$ in time and $\mathcal{O}(V)$ in space). In this case, the nodes of the graph are tasks, and the edges are orderings. So that's $M*(t_m+e_m)$, where e_m is the number of "edges" in the new method, and has an upper bound of e_m to find a topological sort for the sub-tasks of every method. That's approximately $\mathcal{O}(|M|*|T|)$, since both t_m and t_e are upper bounded by T, such that $T = T_p+T_C$.

So in total the main algorithm takes:
$(|M|*|F|*|T|)) + (|M|*|F|*|T|^2) + (|M|*|T|^2*c) + (|M|*|T|))$

The number of times we need to remove an edge in a cycle, c, has an upper limit of $|F|$. Adding an additional ordering (which may cause a cycle) requires interaction, and removing the aforementioned additional ordering will remove the cycle. Thus the maximum run-time of this algorithm is in $\mathcal{O}(M*F*T^2)$, i.e. quadratic time at most.

3.3 Theoretical properties

Theorem 2. *Given a POHTN planning problem P and TOHTN problem P' obtained from P by using Algorithm 1 then the solution set of P' is a not necessarily strict subset of that of P.*

Proof. The new desired orderings for a method include all of the orderings already required by the method originally. The algorithm then turns the tasks and new desired orderings between them into a directed graph, and the new ordering is produced by performing a topological sort on the nodes of that graph. This means we do not modify the sub-tasks a method produces, just the ordering between them, so the set of plans from the totally ordered method is just a subset of the plans possible from the partially ordered one. Any solution to the linearized problem is then obviously a solution to the original problem. $\square$

Theorem 3. *Given a POHTN planning problem P and TOHTN problem P' obtained from P by using Algorithm 1 then the solution set of P' may be empty.*

Proof. This algorithm linearizes all the methods to be totally ordered. Since sub-tasks inherit the orderings of their parents, it's impossible to preserve a solution that requires the interleaving of sub-tasks if their respective parents that are already ordered with respect to each other. This proves that the algorithm can remove some possible action sequences, assuming the original domain was not already totally ordered. Consider the simple example problem:

$$\begin{aligned} F &= \{a, b, c\} \\ N_p &= \{A, B, C\} \\ N_c &= \{AC, T_I\} \\ M &= \{(T_I, \{AC, B\}), \\ &\quad (AC, \{A, C\})\} \\ S_I &= \{a\} \end{aligned}$$

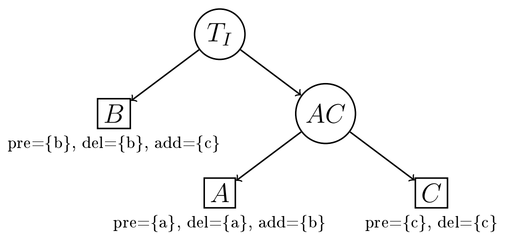


Fig. 4: Diagram showing an example problem and its decomposition.

$$\begin{tikzcd} S_I \arrow[r, "a"] & \boxed{A} \arrow[r, "b"] & \boxed{B} \arrow[r, "c"] & \boxed{C} \end{tikzcd}$$

Fig. 6: The only possible solution A, B, C for E.g. 1, requires the children of AC and B to be interleaved, meaning we cannot impose an order between them

The only decomposition for this problem results in the set of un-ordered actions $\{A, B, C\}$. If we consider that for the 3! linearizations of this set, the only executable one is A, B, C. This is impossible to achieve by linearized methods, since ordering either AB before C or C before AB will exclude the solution. Even if we were to produce $k!$ methods for each partially ordered method with k unordered sub-tasks, we would not be able to preserve any solution for this problem.

This proves that the algorithm can remove all solutions, so Algorithm 1 is not complete. Note that incompleteness already follows from complexity theory as it's theoretically impossible to turn an arbitrary undecideable problem into a decidable one. Specifically, solutions that require interleaving of sub-tasks will not be preserved, as the example above demonstrates. □

We will now see that our algorithm preserves at least one solution as long as Algorithm 1 does not have to break cycles.

Theorem 4. *Given a POHTN planning problem P and TOHTN problem P' obtained from P by using Algorithm 1, if Algorithm 1 did not have to cycle-break, then if the solution set of P is non-empty then the solution set of P' will be non-empty as well.*

Proof. Assume that there exists a solution in the PO domain. By using the same decomposition sequence in the linearized domain, we can produce the same set of actions as in the PO solution, but with a linearization of the actions decided by the linearized domain. Assume this sequence is $(a_0, a_1, ..., a_n)$. We then prove by induction over the sequence $(a_0, a_1, ..., a_n)$ that it is executable. If $(a_0, a_1, ..., a_n)$ is not executable, that means there exists some action $a_k, 0 < k < n$ that is not executable in the corresponding state. The action a_k could only be non-executable, if one or more of its preconditions was not met. Assume one of these unmet preconditions is for existence of the state variable A. The action a_k must be executable in some linearization of $\{a_0, ..., a_n\}$, as we assumed it was a PO solution. So there must exist an action a_i, $0 < i < n$, that will add A. Actions a_0 and a_k must have a shared parent p in a Task Decomposition Tree. So p has subtasks t_0 and t_k that are parents of a_0 and a_k respectively.

The linearization of this method would have drawn an ordering (t_i, t_k) due to the way the algorithm defines $prec^*, add^*$ etc. We are assuming that all methods linearized without conflict, so (t_i, t_k) should not be required. This safely enforces (a_k, a_0) ordering in the final TO plan, meaning a_0 is not the first action in the resulting total order imposed by the algorithm. In other words, if a_k's precondition could be met by any action a_i, a_i would be ordered in front of it.

If a_i does not exist then a_k can never be executed for any linearization of $\{a_0, ...a_n\}$, contradicting the assumption that this was a PO solution. Since each action in the solution is executable, the entire sequence is executable linearization of actions produced by decomposition of initial task, i.e. the solution. □

There are several levels of "completeness" possible.
- all solutions remain
- at least one solution remains
- all optimal solutions remain
- at least one optimal solution remains

Given what's proven in Theorem 3 and 4, Algorithm 1 guarantees at least one solution remains, if no cycle-breaking is needed. If cycle-breaking is needed, Algorithm 1 makes no completeness guarantees at all – it may remove all possible solutions. Finally, Algorithm 1 makes no decisions on any metric of optimality, so obviously cannot guarantee completeness that any optimal solutions remain.

Theorem 5. *For any problem P that satisfies the criterion for linearization without cycle-breaking, the problem P' obtained from applying the algorithm to P forms a new class of decidable problems.*

Proof. A problem P' obtained from applying Algorithm 1 to any arbitrary P is a totally ordered problem, which are known to be decidable [1]. □

4 Empirical Evaluation

To prove that our technique is beneficial, we conduct a standard empirical evaluation on PO domains and compare the runtime of a state-of-the-art planning system on the original problem vs. the transformed problem. All problems are taken from two benchmark sets prepared for 2020 IPC. The first set[1] was used in the partially ordered track in the 2020 IPC benchmark, while the second set[2] went un-used for various reasons. However, the second set still contains valid POHTN problems, so we include it here as well. Each problem has the standard IPC time limit of 30 minutes. Where grounding and pre-processing is needed, the time to do this is included as part of the 'solving' time required. Since the pre-processing may *potentially* remove all solutions, where the planning system can prove that the pre-processing renders a problem unsolvable, we use the remaining time to solve the original problem.

Hardware and Planning Systems The empirical evaluation was conducted on a machine with 30GiB of memory and 4 vCPUs, each with 2 GiB RAM, for a total of 8 GiB RAM. We use the $panda_\pi$ solver [10] [12] [11] on the configuration of Greedy best first search with visited lists. We try this search with 4 different RC-heuristics: FF [8], Add [3], Filter [11] and Landmark-cut [7].

We compare the performance of these 4 planner settings on the original problem and the pre-processed problem. We also try the Lilotane planner [16], which is specialised for TOHTN problems.

4.1 Results

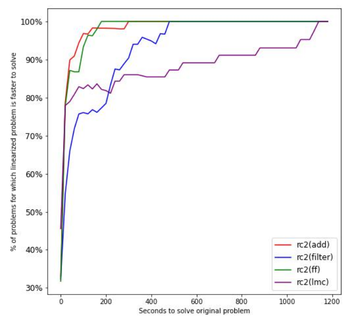


Fig. 7: Percentage of problems for which linearized problems is faster to solve, for problems that took at least x seconds to solve, with x being the value on the x-axis.

[1] https://github.com/panda-planner-dev/ipc2020-domains/tree/master/partial-order

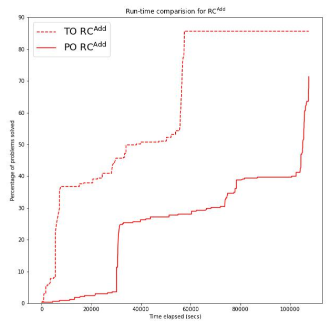
[2] https://github.com/panda-planner-dev/domains/tree/master/partial-order

		RC^{add}		RC^{Filter}		RC^{FF}		$RC^{LM\text{-}Cut}$		Lilotane
	max	PO	TO	PO	TO	PO	TO	PO	TO	
Barman-BDI	1	0.08	**0.4**	0.07	**0.34**	0.07	**0.36**	0.05	**0.22**	**0.66**
Monroe Fully Observ. (2)	1	**0.56**	0.45	**0.31**	0.3	**0.46**	0.41	**0.22**	0.18	0.07
Monroe Part. Observ. (2)	1	**0.31**	0.25	**0.13**	0.11	**0.31**	0.26	**0.17**	0.14	0.0
PCP (17)	1	**0.82**	**0.82**	**0.82**	**0.82**	**0.82**	**0.82**	**0.82**	**0.82**	0.0
Rover	1	0.29	**0.95**	0.14	**0.52**	0.2	**0.78**	0.16	**0.48**	**0.98**
Satellite	1	0.91	**1.0**	0.76	**1.0**	0.99	**1.0**	0.89	**0.99**	**1.0**
SmartPhone (1)	1	**0.71**	**0.71**	0.69	**0.71**	**0.71**	**0.71**	**0.71**	**0.71**	**0.71**
Transport	1	0.24	**0.61**	0.04	**0.05**	0.27	**0.32**	0.12	**0.2**	**0.71**
UM-Translog (1)	1	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	0.95
Woodworking (2)	1	0.38	**0.58**	0.2	**0.41**	0.36	**0.57**	0.27	**0.39**	0.47
Monroe	1	**0.77**	0.69	**0.5**	0.47	**0.75**	0.71	**0.53**	**0.53**	0.46
SmartPhone (1)	1	**0.71**	**0.71**	0.69	**0.71**	**0.71**	**0.71**	**0.71**	**0.71**	**0.71**
Zenotravel	1	**1.0**	**1.0**	0.63	**1.0**	**1.0**	**1.0**	0.83	**1.0**	**1.0**
Total IPC score	13	7.8	**9.2**	6.0	**7.5**	7.7	**8.7**	6.5	**7.4**	7.7

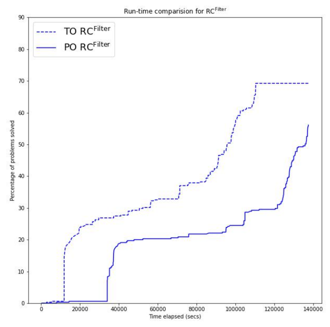
Table 1: IPC score, with and without pre-processing, for all planners. If any problems in that domain were proven unsolvable by TO, a number in brackets beside domain name shows how many.

		RC^{add}		RC^{Filter}		RC^{FF}		$RC^{LM\text{-}Cut}$		Lilotane
	max	PO	TO	PO	TO	PO	TO	PO	TO	
Barman-BDI	20	3	**10**	3	**10**	3	**10**	2	**9**	**16**
Monroe Fully Observ. (2)	25	**25**	**25**	18	**25**	22	**25**	15	**16**	6
Monroe Part. Observ. (2)	24	**14**	**14**	**7**	**7**	14	**15**	**10**	**10**	0
PCP (17)	17	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**	0
Rover	20	6	**20**	4	**14**	4	**19**	4	**14**	**20**
Satellite	25	24	**25**	22	**25**	**25**	**25**	24	**25**	**25**
SmartPhone (1)	7	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
Transport	40	12	**28**	**2**	**2**	13	**14**	7	**12**	**31**
UM-Translog (1)	22	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	21
Woodworking (2)	30	13	**19**	7	**15**	12	**20**	9	**15**	15
Monroe	100	96	**100**	79	**88**	92	**100**	81	**90**	83
SmartPhone (1)	7	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
Zenotravel	5	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
Coverage	342	244	**292**	193	**237**	236	**279**	203	**242**	232
Norm. coverage	13	8.94	**10.67**	7.57	**9.17**	8.71	**10.34**	7.81	**9.16**	8.53

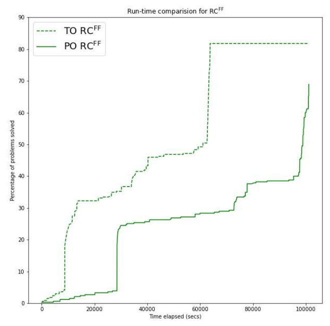
Table 2: Coverage, with and without pre-processing, for all planners. If any problems in that domain were proven unsolvable by TO, a number in brackets beside domain name shows how many.



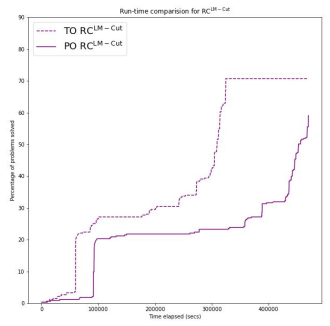
(a) Using heuristic RC$^{\text{add}}$



(b) Using heuristic RC$^{\text{Filter}}$



(c) Using heuristic RC$^{\text{FF}}$



(d) Using heuristic RC$^{\text{LM-Cut}}$

Fig. 8: Comparison of time to solve PO and TO problems, for each $panda_{\pi}$ setting

4.2 Analysis

Table 1 and 2 shows that the IPC score and coverage when pre-processing is overall significantly better than when not using pre-processing.

However in some domains there is minimal gain, if any at all. This seems to be because some domains are dominated by small problems. E.g. 14 of the 17 PCP problems take less than 0.5 seconds to solve. For very small problems, the pre-processing still takes time, but very little improvement can be gained in the actual solving time. In fact, a second of pre-processing time for a problem that only takes a few seconds to solve will worsen IPC score.

On the other hand, big problems experience significant improvement, enough to make problems solvable within the time limit where they were previously too memory or time intensive. We can see that difference in domains like Rover and Barman-BDI, where there is significant space to improve on coverage/speed.

Table 4 and Table 3 summarises the results of our empirical evaluation on the IPC 2020 benchmarks. The "unsolvable" columns refer to problems that

		Solved	Out of Memory	Timeout	Unsolvable
RC^{add}	TO	287	27	21	0
	PO	239	65	31	0
	Either	287	72	40	0
RC^{Filter}	TO	232	85	18	0
	PO	188	115	32	0
	Either	232	125	35	0
RC^{FF}	TO	274	41	20	0
	PO	231	76	28	0
	Either	274	85	37	0
$RC^{LM\text{-}Cut}$	TO	237	6	92	0
	PO	198	11	126	0
	Either	237	12	126	0

Table 3: With re-run policy

		Solved	Out of Memory	Timeout	Unsolvable
RC^{add}	TO	266	24	20	25
	PO	239	65	31	0
	Either	287	72	40	25
RC^{Filter}	TO	212	80	18	25
	PO	188	115	32	0
	Either	232	125	35	25
RC^{FF}	TO	253	38	19	25
	PO	231	76	28	0
	Either	274	85	37	25
$RC^{LM\text{-}Cut}$	TO	217	2	91	25
	PO	198	11	126	0
	Either	237	12	126	25
Lilotane	TO	227	108	0	0

Table 4: Without re-run policy

the solver determined to have no solutions. Lilotane [16], the runner-up for the totally ordered track in the 2020 IPC competition, cannot prove that a problem is unsolvable, and thus can't use a re-run policy, unlike the $panda_\pi$ planner. Note that Lilotane wants a lifted representation, but a grounded representation was used here, to its disadvantage.

The PCP domain in particular was rendered unsolvable for all instances. In Table 1 and Table 2, PCP^{17} indicates that 17 PCP problems were unsolvable. So all 14 problems "solved" in the TO context was from re-running the planner on the original problem using the remaining time. Incidentally, the exact same problems were proven unsolvable for all planners.

The PCP domain defines the post-correspondence problem, which is known to be undecidable. Totally ordered HTN planning problems are always decidable [9], meaning it's a direct consequence that these problem instances are unsolvable without task interleaving. In this specific case, the PCP problems designed for the IPC benchmark are known to be unsolvable without interleaving [13]. Table 3 shows the results on the default setting, when allowing the planner to try solving the original problem as a back-up plan when the linearized problem fails. Lilotane is excluded from Table 3 as it cannot prove a problem is unsolvable, and so cannot use the re-run policy. instead. Table 4 shows results without the default back-up strategy.

From Table 3 and 4 we can see that this processing means that many more problems can be solved in the time limit, where previously they were too computation-intensive, and very few problems (primarily problems for which interleaving is required for all solutions) become unsolvable due to the transformation. Despite the fact that none of the instances can be linearized without cycle-breaking, only 25/274 of linearized problems are unsolvable, 17 of which are from the PCP domain, where a solvable linearization does not exist.

5 Conclusion

Though it's impressive performance on a range of domains is good news, the success of this approach on the IPC benchmark ultimately hinges on the relative lack of unsolvable problems – e.g. PCP problems in the ICAPS benchmark. When the pre-processed domain eliminates all solutions, significant time is wasted in proving this. Fortunately, the IPC benchmark covers a wide variety of domains, so the performance of the pre-processing is hopefully indicative of good performance in other problems as well. Ultimately, it's also an undecidable problem to detect when a problem cannot be converted to TOHTN representation (as that would be solving whether the problem is undecidable), so no perfect solution exists.

6 Further Work

Given that HyperTensioN [14] and Lilotane [16], planning systems that work on *lifted* totally ordered problems, and rely significantly on lifted input for their efficiency, it would be of interest to generalise this technique to produce lifted domains, so that HyperTensioN and Lilotane can solve the linearized problem more efficiently. For another, the success of this solution was, as stated before, in part due to the relatively few undecidable problems in the benchmark set. A heuristic to decide whether or not to pre-process, so that we can reduce the case where the planning system attempts to solve an unsolvable problem might allow this procedure to generalise better in less favourable circumstances.

References

1. Alford, R., Bercher, P., Aha, D.: Tight bounds for HTN planning. In: Proceedings of the 25th International Conference on Automated Planning and Scheduling, ICAPS 2015. pp. 7–15. AAAI Press (2015)

2. Bercher, P., Alford, R., Höller, D.: A survey on hierarchical planning – one abstract idea, many concrete realizations. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, ICJAI 2019. pp. 6267–6275 (2019)
3. Bonet, B., Geffner, H.: Planning as heuristic search. In: Artificial Intelligence. vol. 129, pp. 129(1–2):5–33 (2001)
4. Erol, K., Hendler, J.A., Nau, D.S.: Complexity results for HTN planning. In: Annals of Mathematics and Artificial Intelligence. vol. 18, pp. 69–93 (1996)
5. Geier, T., Bercher, P.: On the decidability of HTN planning with task insertion. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011. pp. 1955–1961. AAAI Press (2011)
6. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning – Theory and Practice. Elsevier (2004)
7. Helmert, M., Domshlak, C.: Landmarks, critical paths and abstractions: What's the difference anyway? In: Proc. of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009. pp. 162–169. AAAI Press (2009)
8. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. In: Journal of Artificial Intelligence Research, 14:253–302 (2001)
9. Höller, D., Behnke, G., Bercher, P., Biundo, S.: Language classification of hierarchical planning problems. In: Proceedings of the 21st European Conference on Artificial Intelligence, ECAI 2014. pp. 447–452. IOS Press (2014)
10. Höller, D., Behnke, G., Biundo, S.: HTN planning as heuristic progression search. In: Journal of Artificial Intelligence Research, JAIR 2020. vol. 67, pp. 835–880 (2020)
11. Höller, D., Bercher, P., Behnke, G., Biundo, S.: A generic method to guide HTN progression search with classical heuristics. In: Proceedings of the 28th International Conference on Automated Planning and Scheduling, ICAPS 2018. pp. 114–122. AAAI Press (2018)
12. Höller, D., Bercher, P., Behnke, G., Biundo, S.: On guiding search in HTN planning with classical planning heuristics. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019. pp. 6171–6175. AAAI Press (2019)
13. Höller, D., Lin, S., Erol, K., Bercher, P.: From PCP to HTN planning through CFGs. In: Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track, IPC 2020. pp. 24–25 (2021)
14. Magnaguagno, M.C., Meneguzzi, F., de Silva, L.: Hypertension: A three-stage compiler for planning. In: Proceedings of the 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track. pp. 5–8 (2021)
15. Olz, C., Biundo, S., Bercher, P.: Revealing hidden preconditions and effects of compound HTN planning tasks – a complexity analysis. In: Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021. pp. 11903–11912 (2021)
16. Schreiber, D.: Lifted logic for task networks: TOHTN planner lilotane in the IPC 2020. In: Journal of Artificial Intelligence Research, JAIR 2021. vol. 70, pp. 1117–1181 (2021)
17. de Silva, L., Sardina, S., Padgham, L.: Summary information for reasoning about hierarchical plans. In: Proceedings of the Twenty-Second European Conference on Artificial Intelligence, ECAI 2016. pp. 1300–1308. IOS Press (2016)