Redundant Decompositions in PO HTN Domains: Goto Considered Harmful

Roland Godet[1,2], Arthur Bit-Monnot[1], Charles Lesire-Cabaniols[2]

[1]LAAS-CNRS, University of Toulouse, INSA, Toulouse, France
[2]ONERA/DTIS, University of Toulouse, France
roland.godet@laas.fr, arthur.bit-monnot@laas.fr, charles.lesire@onera.fr

Abstract

HTN planning is a widely used approach for solving planning problems by breaking them down into smaller sub-problems. This approach is often motivated by the ability to add constraints between tasks, which can guide the search towards a solution and improve performance by reducing the search space. In this paper, we identify a common pattern in PO HTN planning that can lead to a pathological explosion of the search space, resulting in a significant decrease in computational performance. However, this is not a fatal issue. Alternative HTN models can be used to reduce the search space. We propose two models that maintain the expressiveness of the original problem while reducing the number of possible decompositions. Our results demonstrate improved computational performance on IPC benchmarks.

Introduction

Task planning is a fundamental problem in Artificial Intelligence, with applications in robotics, logistics, and many other domains. The problem consists of finding a sequence of actions that completes a given goal, while respecting a set of constraints.

One of the approaches to planning is Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1994), where actions are hierarchically organized into tasks, which can be refined into subtasks or actions, and so on. This hierarchical structure allows the problem to be described at various levels of abstraction, ranging from highly abstract tasks to directly executable actions.

One common motivation for using HTN planning is the promise of increased performance as the hierarchy is expected to restrict the search space and guide the planner towards a solution. In the case of Partial Order (PO) HTN planning, where several tasks may interact in the achievement of their respective goals, we show that on the contrary, the hierarchy can be extremely detrimental to the search.

After a brief introduction to the HTN formalism, this paper identifies a pattern that is ubiquitous in PO HTN planning benchmarks, and that leads an explosion of the search space of PO HTN planners. While there is no general approach to solve this issue, we propose new models that reduce the number of possible decompositions, and show that they improve the computational performances on the International Planning Competition (IPC) 2020 HTN tracks.

HTN Planning

This section provides a brief introduction to the HTN planning problem as described by Höller et al. (2020).

An HTN planning problem can be notably described using the HDDL language (Höller et al. 2020), an extension of PDDL (McDermott et al. 1998), or the ANML language (Smith, Cushing, and Frank 2008).

Assume that $\mathcal{L} = (P, T, V, C)$ is a quantifier- and function-free first order predicate logic. T is finite set of type symbols. C is a finite set of typed constants. V is a finite set of typed variables. P is a finite set of predicate symbols, each associated to a list of parameter variables from V.

Definition 1 (State)**.** A state is the representation of the world at a given time, defined by a ground (variable-free) conjunction of literals over $\mathcal{L}$. The set of all possible states is denoted by S.

Definition 2 (Primitive Task)**.** A primitive task (or action) is an operation that can be executed directly, defined by the tuple $a = (name, pre, eff)$ where:

- *name* is its unique task name, a first-order atom such as `move(s, d)` consisting of the action name followed by parameters.
- *pre* is its precondition, a conjunction of first-order literals over $\mathcal{L}$.
- *eff* is its effect, a conjunction of first-order literals over $\mathcal{L}$. We split it into positive (eff^+) and negative (eff^-) effects.

Remark. We also refer to *pre* and *eff* as *pre(a)* and *eff(a)* when referring to a specific action a. All variables used in *pre(a)* and *eff(a)* must be parameters of the action. Finally, an action is said *ground* if all its parameters are constants from C.

Definition 3 (Executable Action)**.** Given a state $s \in S$, a ground primitive task a is executable in s if and only if its precondition is satisfied by s:

$$\xi(s, a) = s \models pre(a) \tag{1}$$

Definition 4 (State Transition)**.** Given a state s and an executable action a, the state transition function $\gamma(s, a)$ is the result of executing the action a in s. It is defined by the following formula:

$$\gamma(s, a) = \big(s \setminus \mathit{eff}^-(a)\big) \cup \mathit{eff}^+(a) \tag{2}$$

Remark. The extension of ξ and γ to a sequence of actions are defined recursively by:

$$\begin{cases} \xi(s, \langle a_1, \ldots, a_n\rangle) = \xi(\gamma(s, a_1), \langle a_2, \ldots, a_n\rangle) \\ \gamma(s, \langle a_1, \ldots, a_n\rangle) = \gamma(\gamma(s, a_1), \langle a_2, \ldots, a_n\rangle) \end{cases} \tag{3}$$

Definition 5 (Compound Task)**.** A compound task (or abstract task) is simply a task name, *i.e.*, a first-order atom such as `goto(p)` consisting of the actual task name followed by parameters.

Definition 6 (Task Network)**.** A task network over a set of task names X is a tuple $tn = (I, \prec, \alpha, C)$ where:

- I is a possibly empty set of task identifiers. They are used to distinguishing between tasks that occur multiple times in the task network.
- $\prec$ is a strict partial order over I.
- $\alpha : I \to X$ maps each task identifier to a task name.
- C is a set of constraints over the task parameters.

Remark. For easier comprehension, we will refer to a task network without constraints (*i.e.*, $C = \emptyset$) and with total ordered tasks as the set $tn = \{t_1 \prec \ldots \prec t_n\}$.

Definition 7 (Method)**.** A decomposition method represents a way to achieve a compound task. It is defined by the tuple $m = (c, tn)$ where c is the compound task name and tn is a task network describing the subtasks needed to achieve the compound task.

Definition 8 (Decomposition)**.** Decomposition is the process of replacing a compound task of a task network by another task network. Given a decomposition method $m = (c, (I_m, \prec_m, \alpha_m))$ and a task network $tn_1 = (I_1, \prec_1, \alpha_1)$ such that $I_m \cap I_1 = \emptyset$ (that can be done by renaming), the task network $tn_2 = (I_2, \prec_2, \alpha_2)$ is a decomposition of a task identifier $i \in I_1$ by m if and only if:

$$\begin{cases} \alpha_1(i) = c \\ \quad I_2 = (I_1 \setminus \{i\}) \cup I_m \\ \quad \prec_2 = (\prec_1 \cup \prec_m \\ \qquad \cup \{(i_1, i_2) \in I_1 \times I_m \mid (i_1, i) \in \prec_1\} \\ \qquad \cup \{(i_1, i_2) \in I_m \times I_1 \mid (i, i_2) \in \prec_1\}) \\ \qquad \setminus \{(i', i'') \in I_1 \times I_1 \mid i' = i \text{ or } i'' = i\} \\ \quad \alpha_2 = (\alpha_1 \cup \alpha_m) \setminus \{(i, c)\} \end{cases} \tag{4}$$

Definition 9 (Executable Task Network)**.** Given a state s and a ground task network tn, the task network tn is executable in s if and only if:

- the constraints of C are respected.
- there exists a sequence $\langle i_1, \ldots, i_n\rangle$ of its task identifiers, with $n = |I|$, respecting $\prec$ such that $\langle \alpha(i_1), \ldots, \alpha(i_n)\rangle$ is executable in s.

Definition 10 (Planning Domain)**.** A planning domain is a tuple $\mathcal{D} = (\mathcal{L}, T_P, T_C, M)$ where:

- $\mathcal{L}$ is a predicate logic.
- T_P and T_C are sets of primitive and compound tasks.
- M is a set of methods with compound tasks from T_C and task networks over the names $T_P \cup T_C$.

Definition 11 (Planning Problem)**.** A planning problem is a tuple $\mathcal{P} = (\mathcal{D}, s_I, tn_I, g)$ where:

- $\mathcal{D}$ is a planning domain.
- $s_I \in S$ is the initial state, a ground conjunction of positive literals over the predicates.
- tn_I is the initial task network.
- g is the goal, a first-order formula over the predicates.

Definition 12 (Solution)**.** Given a planning problem $\mathcal{P} = (\mathcal{D}, s_I, tn_I, g)$, where $\mathcal{D} = (\mathcal{L}, T_P, T_C, M)$, a task network $tn_S = (I_S, \prec_S, \alpha_S)$ is a solution of $\mathcal{P}$ if and only if:

- there is a sequence of decompositions from tn_I to $tn = (I, \prec, \alpha)$, such that $I = I_S$, $\prec \subseteq \prec_S$, and $\alpha = \alpha_S$.
- tn_S is executable in s_I and its execution leads to a state s such that $s \models g$.

Motivating Example

Let us consider a simple navigation problem consisting of a truck that must go to a given position. The aim for the truck is to go from the position `p1` to the position `p5` using the roads defined in Figure 1.

Problem Formalization

Let us first correctly formalize this problem.

Predicate Logic The predicate logic is defined by the tuple $\mathcal{L} = (P, T, V, C)$ where:

- $T = \{\texttt{T}, \texttt{P}\}$, `T` represents a truck and `P` a position.
- C is composed of one truck `t1` and five positions from `p1` to `p5`.
- $P = \{\texttt{at(t, p)}, \texttt{road(s, d)}\}$, with `at` representing that the truck `t` is at the position `p` and `road` the existence of a road between the positions `s` and `d`.
- V is the set of variables appearing in the next defined actions, tasks, and methods.

Primitive Tasks There are two primitive tasks:

- `move(t, s, d)`, that moves the truck `t` from the position `s` to the position `d` if there is a road.
  - *pre(move)* = `at(t, s)` $\wedge$ `road(s, d)`
  - *eff(move)* = `at(t, d)` $\wedge \neg$`at(t, s)`
- `noop(t, d)`, that does nothing and is only applicable if the truck `t` is at the position `d`.
  - *pre(noop)* = `at(t, p)`
  - *eff(noop)* = $\emptyset$.

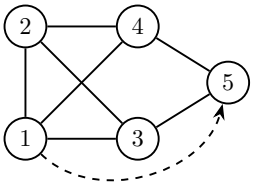


Figure 1: Graph of the navigation problem. The truck can move from one position to another using the roads. It is initially at the position `p1` and must go to the position `p5`.

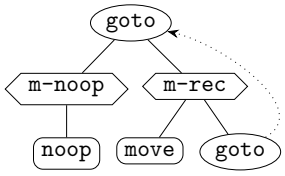


Figure 2: Hierarchical model of the `goto` task. There are two methods to achieve it: (i) do nothing if the truck is already at the given position, (ii) `move` to a nearby position, and then `goto` the given position again.

Compound Tasks & Methods We define a single compound task, `goto(t, d)`, that recursively moves the truck `t` to the position `d`. Two methods can achieve this task as shown in Figure 2:

- do nothing if the truck is already at the given position: `m-noop` = {`noop(t, d)`}.
- move to a nearby position, and then go to the given position again:
  `m-rec` = {`move(t, s, n)` $\prec$ `goto(t, d)`}.

Domain The domain is simply defined by:
$\mathcal{D} = (\mathcal{L}, \{\texttt{move}, \texttt{noop}\}, \{\texttt{goto}\}, \{\texttt{m-rec}, \texttt{m-noop}\})$.

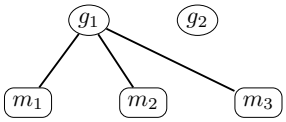
Problem The initial state is defined such that the truck is at the position `p1` and the roads match the graph in Figure 1:
s_I = `at(t1, p1)` $\wedge$ `road(p1, p2)` $\wedge \ldots$
The initial task network is composed of n identical and unordered `goto(t1, p5)` tasks, without constraints. Finally, the problem is defined by $\mathcal{P} = (\mathcal{D}, s_I, tn_I, \emptyset)$.

Pattern Identification

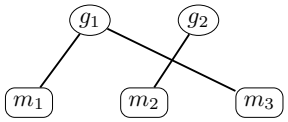
This recursive representation of the `goto` task is common in hierarchical planning community. It is found in many domains, such as *Factory*, *Transport*, or *Minecraft Player* of the HTN track of the IPC.

Consider the problem with three `goto(t1, p5)` in the initial task network with the respective identifiers g_1, g_2, and g_3. In essence this means that three different tasks request the objective of bringing the truck to `p5`.

The shortest solution is composed of two `move` actions m_1 and m_2, for instance going through `p3` with m_1 : `move(t1, p1, p3)` and m_2 : `move(t1, p3, p5)`. Note that we do not explicitly consider `noop` actions in the solution as they are artifact of the hierarchical encoding to break the recursion and could be replaced by method preconditions in the HDDL formalism.



(a) g_1 decomposes all actions.



(b) Decomposition is shared.

Figure 3: Considering two identical `goto` tasks (g_1, g_2), and an optimal solution with three `move` actions. Two possible decompositions of the initial task network that lead to the same solution plan. There are 8 possible decompositions in total.

In order for this $\langle m_1, m_2 \rangle$ action sequence to be considered a solution, it must be decomposable from the initial task network. Intuitively, each of the three (indistinguishable) top level tasks can be decomposed into this sequence. Interestingly, in the absence of ordering constraints, m_1 and m_2 may be decomposed from different top-level tasks. For instance, we may have for instance g_1 moving the truck from `p1` to `p3` and g_3 from `p3` to `p5`.[1] This situation is illustrated in Figure 3.

Let us now consider the number of decomposition paths that lead to this particular optimal solution. The planner will first decompose a `goto` task g_i ($i \in \{1, 2, 3\}$) into the `move` action m_1 and contribute a fresh `goto` task g_4 to the task network. Then, it will decompose a `goto` task g_j ($j \in \{1, 2, 3, 4\} \setminus \{i\}$) into the `move` action m_2 and another `goto` task g_5. Finally, it will decompose all `goto` task g_k ($k \in \{1, 2, 3, 4, 5\} \setminus \{i, j\}$) into the `noop` action.

Because at each step the planner can choose any task to decompose, the number of possible decompositions depends on the number of tasks in the task network. In our example, each `move` action can be decomposed from any of the three `goto` tasks, resulting in $3^2 = 9$ possible decompositions.

This result is trivially generalizable to n `goto` tasks and k `move` actions, resulting in n^k possible decompositions. For instance, in the case of six `goto` tasks and ten `move` actions, the planner must consider $6^{10} = 60466176$ possible decompositions.

HTN planners typically explore the set of possible decompositions of the initial task network until one is found that is both primitive and executable (*i.e.*, a solution). As a result this redundancy of decomposition paths is likely to translate as a redundancy in the search space of HTN planners.

Empirical Tests

To illustrate the impact of this pattern on the computational performance of a planner, we conducted a simple

[1] This only requires the `noop` actions to appear last in the plan, which is not prevented by any ordering constraints.

experiment on the domain of the example.

We used the PandaPi planner (Holler 2023), a state-of-the-art PO HTN planner, on the IPC 2023 HTN track, without timeout on 10 instances. For the i-th instance, we considered i identical `goto(t1, p5)` tasks in the initial task network. For each instance, the shortest plan contains exactly two `move` actions and i `noop` actions.

The results are shown in Figure 4. The five first instances are solved in less than 1 second, while the last instance is solved in 15 minutes.

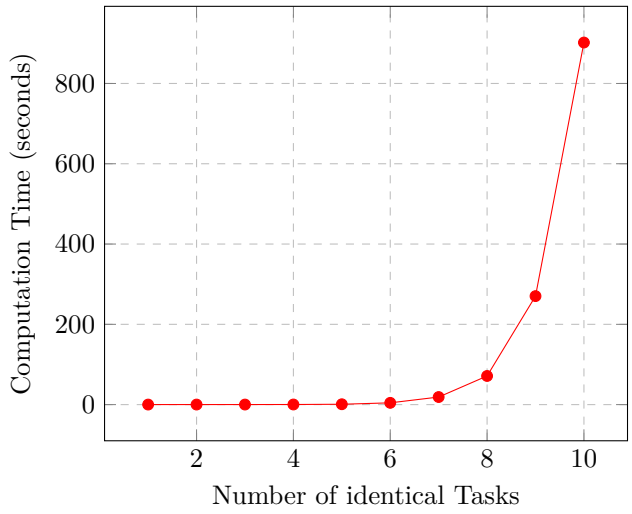


Figure 4: Planning time of PandaPi as a function of the number of identical `goto` tasks in the initial task network. In all cases, the shortest plan requires 2 `move` actions.

Representativity of the Use Case

While this pattern may appear artificial, we argue that is in fact pervasive in PO HTN planning benchmarks. Consider the logistic problem of moving packages from one location to another, as in the *Transport* domain of the IPC 2020 HTN track.

This is generally represented with a `deliver` task whose main method (i) move the truck to the package, (ii) load the package onto the truck, (iii) move the truck to the destination, and (iv) unloads the package at the destination. The first and third steps are typically done with a `goto` task similar to the one we showed.

If we consider the problem of moving three packages initially at location `X` to a location `Y` and assume that the truck has sufficient capacity to hold them all, the optimal plan would involve moving the truck to `X`, loading all packages in `X`, moving the truck to `Y` and unloading package at `Y`. Exactly as in our motivating example, the actions necessary to move the truck from `X` to `Y` could be decomposed from the three `goto` tasks introduced by the step *(iii)*.

This is representative of sharing common steps of the plan among potentially concurrent top level tasks. While the initial task network would likely never involve the same task multiple times, the decomposition of the common steps of the plan would lead to the same issue as the one we identified in our use case.

New Models

As shown in the previous section, the hierarchical model of the `goto` task shown in Figure 2 results in an exponential number of possible decompositions. This negatively impacts the computational performance of the planner, making the problem intractable for large instances.

This section describes two alternative models to the one shown in Figure 2, but with a smaller number of possible decompositions.

Mutex Decomposition

The explosion in the number of possible decompositions is primarily due to potential interference between several tasks for the purpose of producing a sequence of actions, as illustrated in Figure 3b. The idea in this alternative model is to forbid such interference by (i) forcing each `goto` task to lock the truck before producing any `move`, and (ii) only allowing it to release the lock once it has reached its target.

To achieve that, the `goto` task of the previous model is renamed to `goto-exec` and a new `goto(t, d)` compound task is added. The aim of the renamed `goto-exec` task is to force the planner to completely decompose the `goto-exec` task into a sequence of `move` actions that reaches its target `d`. The new `goto` task is used to keep the expressiveness of the domain unchanged.

To ensure that only one `goto-exec` task is decomposed at a time, a mutex predicate `mutex(t)` is added to P. This mutex predicate is manipulated by two new actions, `set-mutex` and `release`:

- `set-mutex(t)` sets the mutex predicate.
  - *pre*(`set-mutex`) = ¬`mutex(t)`
  - *eff*(`set-mutex`) = `mutex(t)`
- `release(t)` releases the mutex predicate.
  - *pre*(`release`) = `mutex(t)`
  - *eff*(`release`) = ¬`mutex(t)`

Finally, the `goto` task can be decomposed by two methods:

- do nothing if the truck is already at the given position: `m-noop` = {`noop(t, d)`}.
- set the mutex predicate, effectively go to the given position, and release the mutex predicate: `m-goto` = {`set-mutex` ≺ `goto-exec` ≺ `release`}.

The final hierarchy of the domain is shown in Figure 5.

The two new actions are used to ensure that the truck will not be moved by another `goto-exec` task while it is already moving. Because a precondition of the `noop` action is for the truck to be at the given position, the

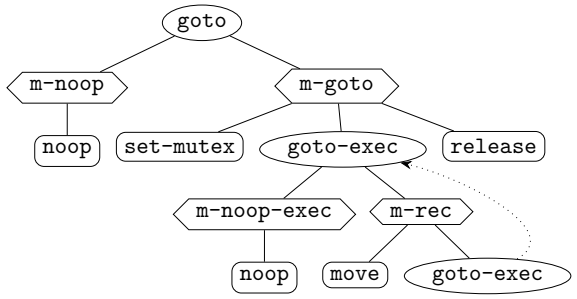
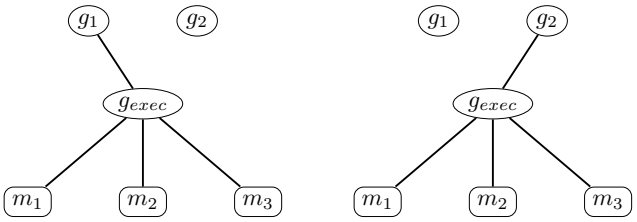


Figure 5: Mutex model of the `goto` task.



(a) g_1 makes the execution. (b) g_2 makes the execution.

Figure 6: Considering two identical `goto` tasks, and an optimal solution with three `move` actions. There are only 2 possible decompositions in total.

mutex will be released only when the truck is effectively at the given position. Therefore, the next `goto` tasks will immediately be decomposed into the `noop` action if it was identical to the previous one.

With this model, the expressiveness of the domain is kept unchanged and, modulo the mutex actions, the set of solutions is the same.

Consider our motivating example of n identical `goto` tasks that should be used to produce a sequence of k `move` actions. Here, the number of possible decompositions is in $\mathcal{O}(n)$: the planner may only choose which of n `goto` tasks to use to produce the full sequence, as illustrated in Figure 6.

(Partial) Task Insertion

As the root of the problem comes from the fact that the `move` actions may be contributed by several concurrent tasks in the initial task network, one solution may be to decouple the introduction of the `move` actions from the top-level objective tasks.

One way to do this would be by adopting the task-insertion variant of HTN planning, where primitive tasks may be introduced independently of any top-level task at arbitrary points of the solution (Alford, Bercher, and Aha 2015). If we were to discard the `goto` task entirely, this would allow the `move` actions to be introduced on-demand to establish the preconditions of actions requiring the truck to be at a given location. Task-insertion however is not without tradeoffs as the freedom of inserting arbitrary tasks in the plan nullifies the ability of HTN models to restrict the set of admissible solutions.

Instead, we propose to mimic the notion of *task-dependency* of FAPE (Bit-Monnot et al. 2020). FAPE distinguishes *task-dependent* actions that may only be introduced through a decomposition and *task-independent* actions that may also be inserted independently of the hierarchy. This enabled FAPE to consider a continuum between generative and HTN planning, where only a subset of the actions are allowed to be the subject of task insertion.

In this model we want to reproduce a similar behavior, where (i) the `move` actions can be inserted arbitrarily, and (ii) a `goto(t, p)` task only imposes a condition that `at(t, p)` holds.

To encode this in a conventional (*i.e.*, without task-insertion or *task-dependency* concepts) HTN model, we introduce a new compound task `free-move(t)` that can be decomposed with two methods:

- `m-move` = {`move(t, s, d)` $\prec$ `free-move(t)`}, that moves the truck between two arbitrary positions `s` and `d` and repeats. Here, `s` and `d` are unconstrained parameters of the method.
- `m-stop-free` = {}, which ends the recursion.

To ensure that the freedom of movement is effective, it is necessary to include the `free-move` task in the initial task network for each truck object. For our example, `free-move(t1)` is added in the initial task network.

To avoid relaxing its post-conditions, the `goto` task is kept in the domain but can only be decomposed by the single method `m-noop`. Because this method contains only the `noop` action, the `goto` task will be decomposed into the `noop` action, effectively doing nothing but checking if the truck is at the required position.

The final hierarchy of the domain is shown in Figure 7.

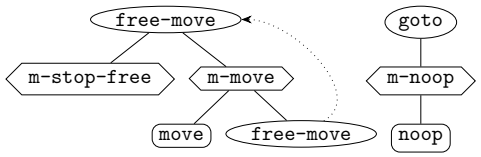


Figure 7: Task insertion model of the `goto` task.

Because each `goto` task is decomposed into a single action `noop`, the number of possible decompositions of n tasks `goto` is constant. Moreover, the number of possible decompositions of the `free-move` task is also constant because it is the only one introducing the sequence of `move` actions needed to achieve the optimal plan and verifying the preconditions of the `goto` tasks. Therefore, the number of possible decompositions for this optimal action sequence is constant and equal to 1.

With this model, the set of solutions differs as the model is strictly more permissive. Any solution of the original model is also a solution of this one, but, because of the freedom of movement, it may be the case that "useless" move actions are inserted in the plan. It should

nevertheless be noted that any optimal (*i.e.*, shortest) plan of the original model is also an optimal plan of this model.

Experiments

To compare the computational performances of the three models, we conducted experiments on our *simple-goto* domain and some domains of the IPC's HTN track. Each domain is duplicated and modified to have four different versions:

- *original*: the original domain.
- *common*: the domain with the common decomposition model, *i.e.*, the one shown in the Motivating Example section.
- *mutex*: the domain with the mutex model.
- *insert*: the domain with the task insertion model.

Domains

We conducted experiments on the following domains, which are available in the IPC's HTN track (except for our example domain). They have been chosen to show the impact of the models on different types of realistic problems.

Goto Simple The goto simple domain is the one described since the beginning of this paper. The *original* and the *common* versions are the same as the one of the motivating example, with an exponential number of possible decompositions. For an instance i ($1 \leq i \leq 30$), the initial task network is composed of $10 * i$ tasks `goto(t1, p5)` and the truck `t1` is initially at `p1`.

Goto Complex The goto complex problem is a more complex version of the goto simple problem. The domain is the same as the goto simple domain, but the initial task network is composed of $10*i$ tasks `goto(t1, p5)`, $10 * i$ tasks `goto(t1, p3)`, for an instance i. It is used to show the impact of nested high-level tasks on the computational performances of the planners.

Factories The *original* version of the factories-simple domain (Sönnichsen and Schreiber 2021) is the one from the IPC's HTN track. It describes the problem of constructing a factory from different resources, each resource needing to be produced from another less-advanced factory. To bring the resources from one factory to another, a truck is used, and its movement is described by the same `goto` task as in the goto simple domain. Therefore, the *common* version is the same as the *original* one (only the *common* version will be displayed in the future results) and the *mutex* and *insert* versions are easily built as described above.

Rovers The *original* version of the rovers domain (Pellier and Fiorino 2021) is the one from the IPC's HTN track. It describes the problem for a set of rovers to navigate and collect data on another planet, before communicating the collected data to the scientists. In this version, the navigation of the rovers is described by a `navigate-abs` task that can be decomposed by three methods: (i) do nothing, (ii) `navigate` to the given position, (iii) `navigate` to an intermediate position, and then `navigate` to the destination. Interestingly, the task is not recursive and the rover can only navigate to a location separated by at most one intermediate position. This is however not an issue because the instances are built such that if the rover needs to go to a location separated by more than one intermediate position, it will need to do another task, *e.g.*, collect data, before going to the final destination. Thus, because the `navigate-abs` is in the task network of several methods, the rovers can accomplish its mission. The *common* version is built by replacing the methods of the original `navigate-abs` task in order to make it recursive. The *mutex* and *insert* versions are then built as described above based on the *common* version.

Transport The *original* version of the transport domain (Behnke, Höller, and Biundo 2018) is the one from the IPC's HTN track. It describes the problem of transporting packages from one location to another using a truck. The truck is capable of carrying multiple packages at once, under a certain limit, and the movement of the truck is described by the `get-to` task that can be decomposed by three methods: (i) do nothing, (ii) `drive` to the given position, (iii) `get-to` an intermediate position, and then `drive` to the destination. In the *common* version, the second method is removed because it is redundant with the third one, and the subtasks order of the third method is changed to `drive` then `get-to` (we use a right recursion instead of a left one). The *mutex* version is build as described above. The *insert* version is obtained by removing all methods of the original `get-to` task except the one that does nothing and adding a `free-drive` as described above.

Planners

We have selected three planners with different resolution strategies to compare the computational performances of the different models. This way we can show the impact of the models on different strategies.

Aries Aries (Bit-Monnot 2023) is a planner transforming *chronicles* (Ghallab, Nau, and Traverso 2004; Godet and Bit-Monnot 2022) into a Constraint Satisfaction Problem (CSP) which is then solved by a specific solver. Because of the recursive nature of the studied domains, the planner needs to instantiate the initial task network until a maximum depth before the generation of the CSP. This depth is set to an initial value and then increased by a fixed step until the planner finds a solution. For the experiments, we are using the version v0.3.3 of the planner[2].

LinearComplex LinearComplex is the winner of the IPC 2023 HTN Partial Order Satisficing track. The main idea of this planner is to first consider the partially

[2]https://github.com/plaans/aries/tree/v0.3.3

ordered task network as a totally ordered one. For the experiments, we are using the winning version of the planner, named `LinearComplex-config-sat-1`.

PandaPi PandaPi (Holler 2023) is a well known planner in the hierarchical planning community. It performs a progression search on the task network, and uses different heuristics to guide the search in the graph. For the experiments, we are using the version for satisficing partial ordered problems of the IPC 2023, which is using the FF heuristic (Hoffmann and Nebel 2001).

Metrics

We are using the following metrics to compare the computational performances of the different models and planners.

Coverage The *Coverage* evaluates the capability of the planner to solve different problem instances in a given domain. It is defined by

$$Cov = \frac{\text{Number of solved instances}}{\text{Total number of instances}} * 100 \qquad (5)$$

Time Score The *Time Score* evaluates the capability of the planner to quickly find a first solution, and matches the one of the IPC agile tracks. The score is computed based on the time t_i in seconds needed to return the first solution of the instance i and the timeout T (120 seconds in our experiments).

$$TS_i = \begin{cases} 1 & \text{if } t_i < 1 \\ 1 - \frac{\log(t_i)}{\log(T)} & \text{otherwise} \end{cases} \qquad (6)$$

Finally, TS is the mean of all TS_i on every instance, multiplied by 100.

Results

The results of the experiments are shown in Table 1.

As expected, we can notice that the *insert* model allows Aries and PandaPi to solve many more instances than the *common* model.

Surprisingly, the *mutex* model is not performing well for the Aries planner, as shown in the Figure 8 for the Rovers domain. This is due to the fact that the planner instantiate the initial task network until a maximum depth before the resolution. In the *common* model, the planner can take one `move` action in each `goto` task, while in the *mutex* model, the planner must take all `move` actions in a single `goto` task. Therefore, Aries needs to go to a deeper depth to find a solution in the *mutex* model than in the *common* one.

For PandaPi, the *mutex* and the *insert* models are performing better than the *common* model. While the improvement of the *mutex* over the *insert* is relatively modest, when the *insert* model exhibits superior performance, the difference is pronounced. This is clearly visible in the Figure 10 for the Transport domain.

LinearComplex The LinearComplex planner behaves differently from the other planners because it first interprets a PO HTN problem as a Total Order (TO) one, therefore prohibiting any interference among `goto` tasks in the *common* model. As a direct consequence, it does not suffer from the redundancies in the search space and the *common* and *mutex* models have similar performance. Moreover, the *insert* model does not perform well for this planner because it may *require* interleaving of tasks to produce a solution, which is not permitted by TO HTN models.

While this choice of interpreting problems as TO HTN ones appears beneficial for coverage, it should be noted that potential high-quality solution are excluded from the resulting search space. For instance in the Transport domain, a truck can carry multiple packages at once. However, the TO HTN projection initially considered by LinearComplex would treat deliveries one after the other, never transporting more than one package at a time.

Left vs Right Recursion A realistic domain which is representative of the pattern we identified is the Transport domain. The results for this domain are shown in the Figure 10 for PandaPi. We can see that the left recursion of the *original* model has a big negative impact on the computational performances of the planner since all instances timed out. Note that this impact is also visible for LinearComplex as shown in Figure 9. Interestingly, it is not the case for Aries which prefers the left recursion of the *original* model. This may be explained by the fact that Aries relies on a plan-space encoding that is naturally backward chaining from the goals towards the initial state. For such planners, left recursion is better suited as it leaves the end of the plan untouched. On the other hand, forward chaining solvers such as PandaPi and LinearComplex perform better with the right recursion of the *common* model.

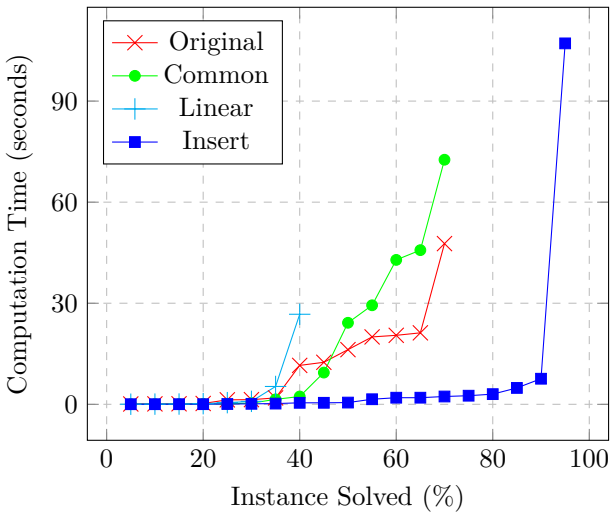


Figure 8: Rovers domain with Aries.

		Aries		PandaPi		LinearComplex	
		Cov	TS	Cov	TS	Cov	TS
Goto Simple	Common	40.00	26.13	0.00	0.00	**100**	**100**
	Mutex	0.00	0.00	**100**	**72.95**	**100**	**100**
	Insert	**100**	**100**	83.33	54.53	76.67	46.55
Goto Complex	Common	26.67	14.07	0.00	0.00	**100**	**98.84**
	Mutex	16.67	8.65	13.33	7.95	**100**	**99.50**
	Insert	**100**	**98.53**	**40.00**	**26.51**	36.67	23.13
Factories	Common	5.00	5.00	**25.00**	**22.68**	**30.00**	**25.09**
	Mutex	5.00	4.22	**25.00**	**22.66**	**30.00**	**24.46**
	Insert	5.00	5.00	25.00	18.98	25.00	19.11
Rovers	Original	70.00	47.09	20.00	10.43	**90.00**	**83.44**
	Common	70.00	47.13	20.00	12.65	80.00	67.52
	Mutex	40.00	34.83	**20.00**	**19.00**	55.00	54.96
	Insert	**95.00**	**81.51**	20.00	17.87	50.00	50.00
Transport	Original	32.50	22.41	0.00	0.00	30.00	24.56
	Common	20.00	14.39	10.00	8.36	**62.50**	**60.25**
	Mutex	17.50	7.60	17.50	11.98	**62.50**	**59.37**
	Insert	**57.50**	**39.24**	**20.00**	**16.28**	55.00	45.52

Table 1: Coverage (Cov) and Time Score (TS) metrics for different planners across different domains and model versions.

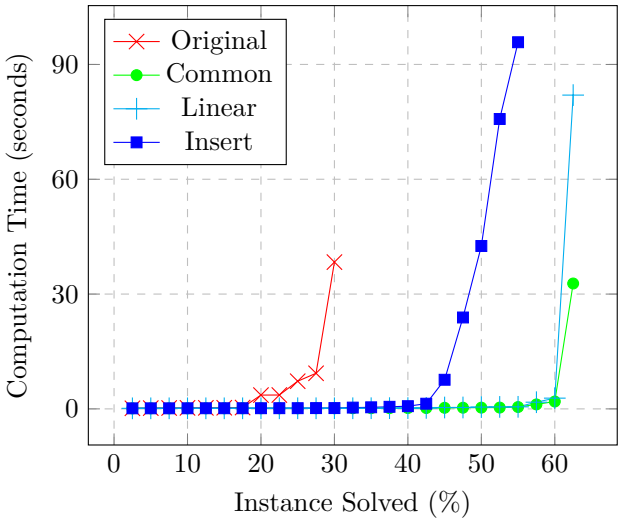


Figure 9: Transport domain with LinearComplex.

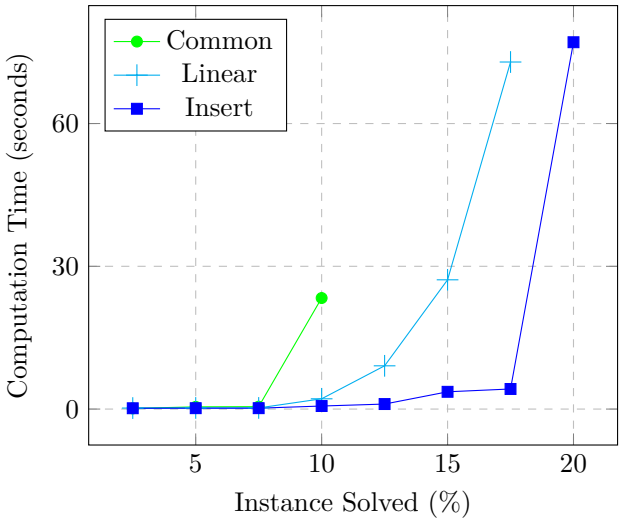
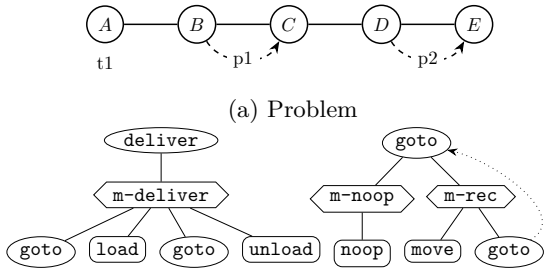


Figure 10: Transport domain with PandaPi. No instance has been solved for the Origin version.

Discussion

The identified pattern is not limited to recursive decomposition of actions as the `move` one, but can also arise from the sharing of common steps of the plan among potentially concurrent top level tasks. This is for instance the case in the Satellite domain of the IPC 2020 HTN track, where the `switch-on` and `calibrate` actions are shared among different tasks. For such cases where a single action may be shared, the impact can be expected to be less dramatic as the number of redundant decomposition would be equal to the number of tasks requiring it.

Finally, let us brought the attention to a false-sharing mechanism that may occur with this pattern. Listing 1 shows a plan with no shared step to sequentially



(a) Problem

(b) Domain Hierarchy

Figure 11: Transport problem with no shared step.

achieve two tasks `deliver(p1, C)` and `deliver(p2, E)`, where `deliver(p, l)` designates the task of delivering the package `p` to the location `l`, as shown in the Figure 11. Even if the plan of the Listing 1 has no shared step, the first and second `move` actions could be attributed to a decomposition of the first `goto` of the second `deliver` task.

Listing 1: Plan with no shared step

```
# Steps 1-4: deliver(p1, C)
move(t1, A, B)
load(p1)
move(t1, B, C)
unload(p1)
# Steps 5-8: deliver(p2, E)
move(t1, C, D)
load(p2)
move(t1, D, E)
unload(p2)
```

Conclusion

In this paper, we have identified a very common pattern in HTN models that can lead to an exponential number of possible decompositions and negatively impact the computational performance of PO HTN planners.

We have proposed two alternative models to the one that leads to this pattern. For native PO HTN planners such as Aries and PandaPi, the model allowing partial task insertion clearly dominates the others and vastly increase the performance on the impacted domains. This suggests that a relaxed HTN models that allow for partial task insertion may be fruitful area for future work.

A notable result is also the drastic performance difference of state-of-the-art planners for different encoding schemes. For instance, unlike PandaPi, Aries favors left-recursive decomposition methods and does not scale up on the *mutex* model. The performance of LinearComplex is highly dependent on the absence of required interleaving between the top-level tasks and thus does not scale on *insert* model. Such issues highlight the fact, at least in PO HTN planning, planner-independent modeling remains a far-fetched goal.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning with Task Insertion. In *International Joint Conference on Artificial Intelligence*.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT - Totally-Ordered Hierarchical Planning Through SAT. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32. ISSN: 2374-3468, 2159-5399 Issue: 1 Journal Abbreviation: AAAI.

Bit-Monnot, A. 2023. Experimenting with Lifted Plan-Space Planning as Scheduling: Aries in the 2023 IPC. In *2023 International Planning Competition at the 33rd International Conference on Automated Planning and Scheduling*. Prague, Czech Republic.

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning. Technical Report arXiv:2010.13121, arXiv. ArXiv:2010.13121 [cs] type: article.

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN Planning: Complexity and Expressivity. *Proceedings of the National Conference on Artificial Intelligence*, 2: 7.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theroy and Practice*. Morgan Kaufmann Publishers Inc. ISBN 978-1-55860-856-6.

Godet, R.; and Bit-Monnot, A. 2022. Chronicles for Representing Hierarchical Planning Problems with Time. In *ICAPS Hierarchical Planning Workshop (HPlan)*. Singapore, Singapore.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Holler, D. 2023. The PANDA Progression System for HTN Planning in the 2023 IPC. In *2023 International Planning Competition at the 33rd International Conference on Automated Planning and Scheduling*. Prague, Czech Republic.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06): 9883–9891.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C. A.; Ram, A.; Veloso, M.; Weld, D. S.; and Wilkins, D. 1998. PDDL-the planning domain definition language.

Pellier, D.; and Fiorino, H. 2021. From Classical to Hierarchical: benchmarks for the HTN Track of the International Planning Competition. *ArXiv*.

Smith, D. E.; Cushing, W.; and Frank, J. 2008. The ANML Language. *KEPS*.

Sönnichsen, M.; and Schreiber, D. 2021. The HTN domain "Factories". In *Proceedings of the 10th International Planning Competition: Planner and domain abstracts – hierarchical task network (HTN) planning track (IPC 2020)*, 45–46.