# Parsing-based Planner for Totally Ordered HTN Planning with Task Insertion

Kristýna Pantůčková
*Charles University*
*Faculty of Mathematics and Physics*
Prague, Czech Republic
pantuckova@ktiml.mff.cuni.cz

Roman Barták
*Charles University*
*Faculty of Mathematics and Physics*
Prague, Czech Republic
bartak@ktiml.mff.cuni.cz

***Abstract*—Hierarchical task network (HTN) planning extends classical planning by incorporating a hierarchy of tasks that gives plans additional structure and speeds up planning. However, it requires that each action be part of some task in the domain model, which makes it less flexible when the task hierarchy does not capture all the possibilities to achieve every task. HTN planning with task insertion (TIHTN planning) extends HTN planning by allowing the insertion of actions outside the hierarchy, thus giving more flexibility to constructing hierarchical plans. TIHTN planning has been proposed as a theoretical concept to show some decidability and complexity results. This paper describes an implemented TIHTN planner for totally ordered domains utilizing top-down grammar parsing.**

***Index Terms*—hierarchical planning, task insertion, parsing**

## I. Introduction

Hierarchical planning [6] extends classical planning by describing the hierarchical structure of tasks, where complex tasks decompose into simpler subtasks until the primitive tasks, executable actions, are reached. Hierarchical planning is often described by the formalism of hierarchical task networks (HTN). The goal of a hierarchical planner is to decompose the initial (goal) task network using task decomposition rules into a sequence of actions, forming an executable plan. The hierarchical task structure helps in two essential aspects:

- it guides the planner towards using the right actions and thus speeding up the planning process,
- it restricts the plans to those somehow expected by human users, making them look more natural.

These properties make hierarchical planning popular, for example, in robotics [12] and computer games [14], where speed and predictability are required. However, the hierarchical structure must be carefully crafted to capture all possible plans for each given task. Specifically, the final plan may contain only actions generated by decomposition of tasks in the domain model. This requirement makes hierarchical planning less flexible than classical planning and, therefore, various hybrid planning approaches have been suggested [7, 13, 20].

Hierarchical planning with task insertion (TIHTN) has been proposed as a formal concept that overcomes the restriction above and helps to prove some results in the decidability and complexity [8]. TIHTN planning extends HTN planning by allowing the insertion of primitive tasks (actions) outside of the task hierarchy. Motivation for TIHTN planning includes defining hierarchical planning problems that could be solved more naturally by a combination of hierarchical and classical planning (instead of incorporating all tasks into a hierarchy) or solving problems where all possible needs of a planning agent have not been taken into account when designing the hierarchical domain model. The TIHTN planning problem was formally defined and its properties were studied [8], but there has been no TIHTN planner implemented.

In this paper, we present a TIHTN planner for totally ordered HTN domain models, which describe problems where each subtask decomposes into an ordered sequence of subtasks. Restriction to totally ordered models should be manageable, as many planning problems can be naturally defined as totally ordered problems – for example, 24 out of the 33 domain models used in the HTN International Planning Competition (IPC) 2020 were totally ordered. The proposed planner is based on top-down grammar parsing, for which we utilize the Earley parser [5] that has already been successfully applied to HTN plan verification [17], HTN plan recognition [15], and HTN plan correction [16].

The paper is organized as follows: first, we recall a formal definition of TIHTN planning; second, we include an overview of related works; then, we describe the proposed TIHTN planner; and finally, we present the empirical evaluation results.

## II. Background on TIHTN planning

Hierarchical planning [6] deals with planning domain models where abstract tasks decompose into subtasks until a sequence of executable primitive tasks (actions) is reached. For simplicity of notation, we will use the grounded representation though the proposed planner is working with the lifted version, where actions and tasks have attributes and there is a finite number of objects represented by constants that substitute the attributes during grounding (see the example below).

We adapt the definition of hierarchical planning with task insertion proposed by Geier and Bercher [8]. A *totally ordered TIHTN planning problem* is defined as $\mathcal{P} = (P, T, A, A^I, R, s_0, tn_0)$, where:

Research is supported by the project 25-18003S of the Czech Science Foundation, by the Johannes Amos Comenius Programme (P JAC) project No. CZ.02.01.01/00/22008/0004605, Natural and anthropogenic georisks, and by SVV project No. 260821.

- $P$ is a finite set of propositions which define planning states ($s \subseteq P$ for each planning state $s$),
- $T$ is a finite set of abstract (decomposable) tasks,
- $A$ is a finite set of all actions (also called primitive tasks),
- $A^I \subseteq A$ is a set of actions which are available for task insertion (i.e., can be inserted regardless of allowed task decompositions),
- $R$ is a finite set of decomposition rules with totally ordered subtasks,
- $s_0 \subseteq P$ is the initial planning state, and
- $tn_0$ is the initial totally ordered hierarchical task network (a sequence of tasks from $T \cup A$).

*Actions* are defined by preconditions and effects:

$$a = (\text{prec}^+(a), \text{prec}^-(a), \text{eff}^+(a), \text{eff}^-(a))$$

where:

- $a$ is applicable in state $s$ if and only if $\text{prec}^+(a) \subseteq s$ and $\text{prec}^-(a) \cap s = \emptyset$, and
- $s' = (s \setminus \text{eff}^-(a)) \cup \text{eff}^+(a)$ is the state resulting from executing $a$ in state $s$.

Possible decompositions of abstract tasks are defined by *decomposition rules*:

$$r = (T_i \rightarrow T_{i_1} ... T_{i_k}, \text{prec}^+(r), \text{prec}^-(r))$$

describing the decomposition of the head task $T_i$ into subtasks $T_{i_1}, ..., T_{i_k}$, where each subtask is either another abstract task or an action. In a *totally ordered domain model*, the subtasks are totally ordered. Rule preconditions are defined similarly to action preconditions: $r$ is applicable in state $s$ if and only if $\text{prec}^+(r) \subseteq s$ and $\text{prec}^-(r) \cap s = \emptyset$, i.e., preconditions of $r$ must be satisfied in the state in which we will execute the first action into which the first subtask ($T_{i_1}$) will be decomposed.

Planning is performed by applying task decompositions or task insertions to a totally ordered hierarchical task network (a sequence of tasks) starting with $tn_0$ and ending with the primitive HTN consisting only of actions.

A *task decomposition* means application of a decomposition rule $r = (T_i \rightarrow T_{i_1} ... T_{i_k}, \text{prec}^+(r), \text{prec}^-(r))$ to an HTN $tn = \langle T_{j_1}, ..., T_{j_m}, T_i, T_{j_{m+1}}, ..., T_{j_n} \rangle$, i.e., replacing task $T_i$ by its subtasks, which results in a new task network $tn' = \langle T_{j_1}, ..., T_{j_m}, T_{i_1}, ..., T_{i_k}, T_{j_{m+1}}, ..., T_{j_n} \rangle$. Note that the rule preconditions are verified in the corresponding state (right before $T_{i_1}$) after the task network becomes primitive and applied to initial state $s_0$.

A *task insertion* corresponds to insertion of an action $a \in A^I$ to some position in a task network $tn = \langle T_{j_1}, ..., T_{j_m}, T_{j_{m+1}}, ..., T_{j_n} \rangle$, which results in a new task network $tn' = \langle T_{j_1}, ..., T_{j_m}, a, T_{j_{m+1}}, ..., T_{j_n} \rangle$.

A *primitive task network* is an HTN that consists only of primitive tasks (actions). A primitive task network $tn = \langle a_1, ..., a_n \rangle$ is *executable in* $s_0$ if and only if:

- $\forall i \in \{1, ..., n\} : s_i = (s_{i-1} \setminus \text{eff}^-(a_i)) \cup \text{eff}^+(a_i)$, and
- $\forall i \in \{1, ..., n\} : a_i$ is applicable in $s_{i-1}$.

A *solution to* $\mathcal{P}$ is a primitive task network $tn^*$, where:

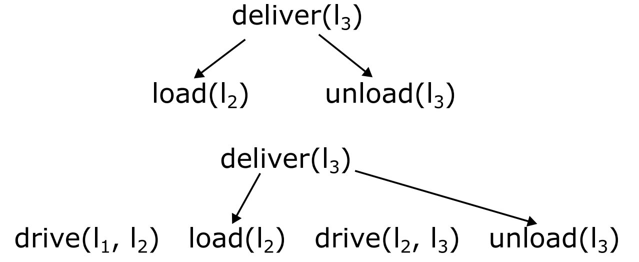


Fig. 1. Example of a primitive (top) and executable HTN (bottom).

- $tn$ is a primitive task network obtained by applying a sequence of the decomposition rules $r_1, ..., r_m$ from $R$ to $tn_0$,
- $tn^*$ was obtained from the task network $tn$ by task insertions,
- $tn^*$ is executable in $s_0$, and
- preconditions of all rules $r_1, ..., r_m$ are satisfied in $tn^*$.

## A. Example

Let us define a simple TIHTN planning problem $\mathcal{P} = (P, T, A, A^I, R, s_0, tn_0)$:

- $P = \{\text{package-location}(l_1), \text{package-location}(l_2), \text{package-location}(l_3), \text{truck-location}(l_1), \text{truck-location}(l_2), \text{truck-location}(l_3), \text{connected}(l_1, l_2), \text{connected}(l_2, l_1), \text{connected}(l_1, l_3), \text{connected}(l_3, l_1), \text{connected}(l_2, l_3), \text{connected}(l_3, l_2), \text{loaded}\}$;
- $T = \{\text{deliver}(l_3)\}$;
- $A = \{\text{load}(l_1), \text{load}(l_2), \text{load}(l_3), \text{unload}(l_1), \text{unload}(l_2), \text{unload}(l_3), \text{drive}(l_1, l_2), \text{drive}(l_2, l_3)\}$, where:
  - $\text{prec}^+(\text{load}(l)) = \{\text{package-location}(l), \text{truck-location}(l)\}$, $\text{prec}^-(\text{load}(l)) = \{\text{loaded}\}$, $\text{eff}^+(\text{load}(l)) = \{\text{loaded}\}$, $\text{eff}^-(\text{load}(l)) = \{\text{package-location}(l)\}$,
  - $\text{prec}^+(\text{unload}(l)) = \{\text{truck-location}(l), \text{loaded}\}$, $\text{prec}^-(\text{unload}(l)) = \emptyset$, $\text{eff}^+(\text{unload}(l)) = \{\text{package-location}(l)\}$, $\text{eff}^-(\text{unload}(l)) = \{\text{loaded}\}$,
  - $\text{prec}^+(\text{drive}(l, l')) = \{\text{truck-location}(l), \text{connected}(l, l')\}$, $\text{prec}^-(\text{drive}(l, l')) = \emptyset$, $\text{eff}^+(\text{drive}(l, l')) = \{\text{truck-location}(l')\}$, $\text{eff}^-(\text{drive}(l, l)) = \{\text{truck-location}(l)\}$;
- $A^I = \{\text{drive}(l_1, l_2), \text{drive}(l_2, l_3)\}$;
- $R = \{\text{deliver}(l_3) \rightarrow \text{load}(l_1)\ \ \text{unload}(l_3), \text{deliver}(l_3) \rightarrow \text{load}(l_2)\ \ \text{unload}(l_3)\}$;
- $s_0 = \{\text{truck-location}(l_1), \text{package-location}(l_2), \text{connected}(l_1, l_2), \text{connected}(l_2, l_3)\}$;
- $tn_0 = \langle \text{deliver}(l_3) \rangle$.

The top picture in Figure 1 displays a primitive task network, which can be found by HTN planning. The bottom picture represents a network executable in the initial state,

which was created by inserting the two *drive* actions to satisfy preconditions of the actions *load* and *unload*.

## III. Related works

Soon after proposing the concept of hierarchical planning, researchers realized that task hierarchies might be incomplete, which reduces the number of possible plans, or too complex to model "all" alternatives and hence difficult to understand. While HTN can encode hierarchical domain knowledge, classical representations are more flexible, e.g., for replanning. Therefore, various hybrid approaches [7, 13] have been proposed that typically extend the refinement planning framework to include HTN planning. Schattenberg and Biundo [19] proposed a unifying framework for hybrid planning and hierarchical scheduling, which is also based on refinement planning. The GoDeL system [20] is using decomposition methods to achieve goals rather than tasks. Hybrid planning was also addressed in Gerevini et al. (2008). However, they approach hybrid planning in a different way. In their problem description, classical and hierarchical planning are clearly separated as some tasks directly trigger classical planning. In contrast, the focus of our paper is creating an executable network from an otherwise valid (but not executable) primitive network. Another approach to handling incomplete task decomposition methods could be to refine the domain model [21].

There exist approaches based on translating HTN planning to classical planning that can theoretically integrate HTN and classical planning [1, 3, 4, 11]. Because HTN planning is strictly more expressive than classical planning [6], translation-based approaches typically require some bound on the size of the task network or the depth of the task decomposition tree. Alford et al. [1] require a bound on the recursion that had to be specified by hand, Alford et al. [3] require tail-recursive domains to calculate the bound, and Behnke et al. [4] also requires a progression bound. Höller [11] uses a different approach that requires plan verification. Our method does not require any bound to be given.

Geier and Bercher [8] presented a formal definition of a propositional TIHTN planning problem. Next, Geier and Bercher proved that the propositional problem of TIHTN plan existence is decidable. Alford et al. [2] proved that propositional totally ordered TIHTN planning is PSPACE-complete, lifted propositional totally ordered TIHTN planning is EXPSPACE-complete, propositional partially ordered TIHTN planning is in NEXPTIME and lifted partially ordered TIHTN planning is in 2-NEXPTIME. These works are rather theoretical, focusing on decidability and complexity results. In our paper, we slightly extend their definition towards more practical applications by allowing preconditions of decomposition rules and insertion of only a subset of actions. Like Geier and Bercher [8], we allow only insertion of primitive tasks (actions) as insertion of an abstract task can be translated into an insertion of primitive tasks. For totally ordered HTN planning, the rule preconditions can be compiled away by introducing dummy actions as the first actions of the methods. However, for TIHTN planning this would change the semantics of the rule precondition to *sometime-before* rather than *exactly-before* since actions from $A^I$ can be inserted between this dummy action and the first subtask of the rule. Original TIHTN planning [8] does not use rule preconditions.

We base our planner on the Earley parser [5], whose application was first proposed for HTN plan recognition [15] and then also for HTN plan verification [17]. Our algorithm is based on the HTN plan correction algorithm [16].

## IV. A totally ordered TIHTN planner

Our totally ordered TIHTN planner is inspired by parsing of *context-free grammars (CFG)*. A CFG is a formal grammar where all rewriting rules can be written in the following way: $A \rightarrow \alpha$, where $A$ is a non-terminal symbol rewritten into $\alpha$, a sequence of terminal and non-terminal symbols. If we omit preconditions of decomposition rules and task insertion, a totally ordered TIHTN planning domain corresponds to a CFG with rewriting rules defined as follows: $T^i \rightarrow T^i_1 ... T^i_k$, where $T^i$ is an abstract task (a non-terminal symbol) and $T^i_1 ... T^i_k$ is a sequence of abstract tasks and actions (terminal symbols).

Our planner approaches a TIHTN planning problem in the following way:

1) A primitive HTN $tn$ is found by applying decomposition rules to the initial task network $tn_0$. Action and rule preconditions are ignored at this point. As this abstraction results in a CFG, we utilize the well-known Earley parser [5] to find the primitive network. The Earley parser is a top-down CFG parser, i.e., it starts from the starting non-terminal symbols (the tasks from the initial task network) and uses rewriting rules (decomposition rules) to reach terminal symbols (actions). Top-down parsing allows us to proceed from the top-level abstract tasks, which are known, to actions, which are unknown. We currently use the Earley parser directly. An advantage of top-down parsing over bottom-up parsing (e.g. CYK) for future research is that hierarchical planning heuristics could be utilized more naturally, which may lead to better efficiency of the planner.
2) We use a classical planner to insert new actions from $A^I$ into $tn$ to obtain an HTN $tn^*$ executable in $s_0$, i.e., to satisfy preconditions of all rules and actions. In this step we use the Fast Downward classical planning system [10]. If an executable network cannot be found for $tn$, we return to step (1) to generate another primitive network.

### A. Using Earley parser for HTN planning

Our algorithm is based on the HTN plan correction algorithm by Pantůčková and Barták (2024) The plan correction approach intends to obtain valid HTN plans from invalid plans by two means of corrections – action deletion (deleting actions from the input plan) and action insertion (inserting new actions into the input plan). To use the plan correction solver for planning, we modify the algorithm by disabling action deletion and specifying the initial task network. We can then obtain

an HTN plan by executing the plan correction solver on an empty input plan. In this section, we provide a brief overview of the resulting algorithm in order to be able to explain the integration of classical planning in the next section.

The Earley parser operates over *parsing states*, which represent partially completed rewriting rules. For example, a parsing state $T^i \rightarrow T^i_1 ... T^i_{j-1} \bullet T^i_j ... T^i_n$ represents a decomposition (rewriting) rule, which decomposes the abstract task $T^i$ into the subtasks $T^i_1, ..., T^i_n$, where the tasks before $\bullet$ ($T^i_1, ..., T^i_{j-1}$) have already been decomposed into actions, while the tasks after $\bullet$ have not been decomposed yet.

First, we generate the top-level parsing state representing the initial HTN. Let $tn_0 = \langle T^0_1, ..., T^0_k \rangle$. Then, the initial parsing state is $S^0 = [T^0 \rightarrow \bullet T^0_1 ... T^0_k]$, where $T^0$ is a dummy starting task.

The Earley parser defines three procedures to process parsing states:

- *predictor* to process parsing states where the first subtask after $\bullet$ is an abstract task,
- *scanner* to process parsing states where the first subtask after $\bullet$ is an action, and
- *completer* to process parsing states where $\bullet$ is at the end.

Let $S = [T^i \rightarrow T^i_1 ... T^i_{j-1} \bullet T^i_j ... T^i_n]$, where $T^i_j$ is an abstract task. Then, $S$ will be processed by *predictor*. $S$ represents a parsing state where subtasks $T^i_1 ... T^i_{j-1}$ have already been decomposed into actions and $T^i_j$ is the next task to be decomposed. To find such decompositions, predictor will search the set of all decomposition rules $R$ to find all rules which decompose task $T^i_j$. For each such rewriting rule $T^i_j \rightarrow T^k_1 ... T^k_m$, predictor will generate a new parsing state $S' = [T^i_j \rightarrow \bullet T^k_1 ... T^k_m]$.

Let $S = [T^i \rightarrow T^i_1 ... T^i_{j-1} \bullet T^i_j \ T^i_{j+1} ... T^i_n]$, where $T^i_j$ is an action. Then, $S$ will be processed by *scanner*. In the original Earley parser, the algorithm would receive a sequence of terminals as an input and scanner would read a next symbol from the input sequence. However, the goal of the planner is to generate such sequence of actions. Therefore, our scanner generates the required action $a = T^i_j \in A$ (where $A$ is a set of all actions) and creates a new state $S' = [T^i \rightarrow T^i_1 ... T^i_{j-1} a \bullet T^i_{j+1} ... T^i_n]$.

Let $S = [T^i \rightarrow T^i_1 ... T^i_n \bullet]$. Then, $S$ will be processed by *completer*. As $S$ represents a decomposition rule where all subtasks have been successfully decomposed, this decomposition rule can be used to decompose task $T^i$ in other rules. Completer will go through all parsing states that have been created until now and for each state where $T^i$ is the next task after $\bullet$, i.e., for each such state $S' = [T^j \rightarrow ... \bullet T^i ...]$, completer will create a new state $S'' = [T^j \rightarrow ... T^i \bullet ...]$. However, if $S''$ has already been created earlier by another completer procedure, we will not create it again.

When the decomposition of the starting parsing state $S^0$ is completed, i.e., when a state $S = [T^0 \rightarrow T^0_1 ... T^0_k \bullet]$ is created, we can find a candidate primitive task network $tn$. A classical planner will then be used in an attempt to create an executable network $tn^*$ from $tn$. If an executable network cannot be created from $tn$, we resume the search as $S$ might be generated again with different decompositions of subtasks, which could lead to an executable HTN.

The original Earley parser progresses from the left to the right on the input sequence and processes parsing states in the order based on indexes of the symbols from the input sequence covered by each parsing state. Our planner instead processes parsing states based on the length of the plan, i.e., on the number of actions generated by decompositions of subtasks of each parsing state. As a result, we prioritize primitive networks consisting of fewer actions, i.e., shorter resulting plans. In the completer, we compute the *cost* of the new state $S''$ as a sum of costs of the subtasks before $\bullet$ (assuming unit action costs). If a subtask has more possible decompositions with different costs, we take its lowest possible cost.

The soundness and completeness of the Earley-based planner is implied by the soundness and completeness of the original Earley parser (if a plan exists, the algorithm will find it). As we ignore preconditions of actions and decomposition rules and we work only with totally ordered domain models, the resulting grammar is a CFG. There is a difference only in the scanner procedure, where we take into account all sequences of actions which can be reached from the top-level tasks, which corresponds to searching multiple input sequences in parallel. As we resume the search if the found $tn$ cannot be extended to an executable $tn^*$, we exhaustively generate all possible primitive networks with the non-decreasing number of actions.

As discussed in the following section, our algorithm can provide a solution with the shortest possible plan (with the lowest number of actions – including the primitive tasks from $tn$ and the actions inserted by the classical planner). If we require an optimal solution, we do not stop the search after the first solution was found, but we continue with the search until there is no state with a *cost* lower than the length of the shortest plan found so far.

### B. Using classical planner for task insertion

During the completer procedure, we remember bidirectional references between each completed subtask and all its possible decompositions (completions). Therefore, when we obtain a candidate primitive network $tn$, we can reconstruct an AND/OR tree rooted in $S^0$, where each subtask of each rule corresponds to an OR-node connected to one or more decomposition rules, as one decomposition rule must be chosen for each subtask, and decomposition rules correspond to AND-nodes connected to its subtasks, as all subtasks of the rule must be decomposed.

To obtain a candidate primitive network $tn$, we explore all decomposition trees which can be created from the AND/OR tree. Obtaining one unambiguous decomposition tree corresponds to choosing one child node at each OR-node and all child nodes at each chosen AND-node.

To create a network $tn^*$ executable in the initial state $s_0$, we need to ensure that preconditions of all actions and decomposition rules can be satisfied by insertion of additional

actions. For this purpose, we define a classical planning problem and we call the classical planner to find a solution.

*1) Classical planning problem:* A *classical planning problem* can be defined as $\mathcal{P}^C = (P^C, A^C, s_0^C, g^+, g^-)$, where:

- $P^C$ is a finite set of propositions,
- $A^C$ is a finite set of actions (defined by preconditions and effects),
- $s_0^C \subseteq P^C$ is the initial state,
- $g^+ \subseteq P^C$ are positive goal conditions, and
- $g^- \subseteq P^C$ are negative goal conditions.

A solution to $\mathcal{P}^C$ is a sequence of actions – a plan – which is reaches the goal $(g^+, g^-)$ and is executable in $s_0$, i.e., $\pi = \langle a_1, ..., a_n \rangle$ is a solution to $\mathcal{P}^C$ if and only if:

- $\forall i (1 \le i \le n) : s_i = (s_{i-1} \setminus \text{eff}^-(a_i)) \cup \text{eff}^+(a_i)$, where $s_0 = s_0^C$,
- $\forall i (1 \le i \le n) : \text{prec}^+(a_i) \subseteq s_{i-1}$ and $\text{prec}^-(a_i) \cap s_{i-1} = \emptyset$, where $s_0 = s_0^C$,
- $g^+ \subseteq s_n$, and
- $g^- \cap s_n = \emptyset$.

Our intention is to define a problem whose solution is a classical plan which contains all actions from $tn$ in the given order and preconditions of all actions and rules are satisfied.

First, we eliminate rule preconditions. As we work with totally ordered domain models, the preconditions of rules can be translated into preconditions of actions. In the decomposition tree, we can find for each chosen decomposition rule the first action into which the first subtask is decomposed (simply by following the left-most branch from the decomposition rule to the leaf). Let $A_{tn}$ be the set of actions in $tn$. For each action $a = (\text{prec}^+(a), \text{prec}^-(a), \text{eff}^+(a), \text{eff}^-(a)) \in A_{tn}$, let $R_a$ be the set of all rules whose first subtask decomposes into a sequence of actions starting by $a$. Then,

$$A'_{tn} = \{(\text{prec}^+(a) \cup \bigcup_{r \in R_a} \text{prec}^+(r), \\ \text{prec}^-(a) \cup \bigcup_{r \in R_a} \text{prec}^-(r), \\ \text{eff}^+(a), \text{eff}^-(a)) \mid a \in A_{tn}\}.$$

Second, we need to enforce that the classical plan contains the actions from $A_{tn}$ in the given order. Here, we use the transformation known from classical plan recognition [18]. For each action $a_i$, we define a new proposition $e_i$, which will be added to the positive effects of this action and to the positive preconditions of the following action. These propositions serve as a token passed between the actions and ensure that each action from $A_{tn}$ appears exactly once in the classical plan. Let $A'_{tn} = \{a_1, ..., a_n\}$, where the order of actions is given by the order of the corresponding leaves in $tn$, $A^I$ be the set of actions that are available for insertion, $e_0$ an additional dummy proposition and $s_0$ the initial state of the TIHTN planning problem. We define a new set of propositions and actions, the initial state and the goal as follows:

$$P^C = P \cup \bigcup_{a_i \in A'_{tn}} \{e_i\} \cup \{e_0\},$$

$$A^C = \{(\text{prec}^+(a_i) \cup \{e_{i-1}\}, \\ \text{prec}^-(a_i), \text{eff}^+(a_i) \cup \{e_i\}, \\ \text{eff}^-(a_i) \cup \{e_{i-1}\}) \mid a_i \in A'_{tn}\} \cup A^I,$$

$$s_0^C = s_0 \cup \{e_0\},$$

$$g^+ = \{e_{|A'_{tn}|}\},$$

$$g^- = \emptyset.$$

Let $\pi = \langle a_1, ..., a_k \rangle$ be the solution to $\mathcal{P}^C$ (the classical plan) and let $f : A_{tn} \to \pi$ be the mapping from the original actions from $tn$ to the modified actions in $\pi$. Let us define a function $g : A^C \to A$:

$$g(a) = \begin{cases} f^{-1}(a) \text{ if } a \notin A^I, \\ \quad a \;\; \text{if } a \in A^I. \end{cases}$$

Then we can create the executable $tn^*$ by adding the new actions from $\pi$ into $tn$: $tn^* = g(\pi)$.

*2) Soundness, completeness, and optimality:* The network obtained $tn^*$ is clearly a correct solution to the TIHTN planning problem $\mathcal{P}$ (assuming that the classical planner is sound). The sequence of actions in $tn^*$ ($A_{tn^*}$) contains $tn$ as a subsequence because the goal $g^+$ can be satisfied if and only if all actions from $A^C \setminus A^I$ were executed. Only actions from $A^I$ are added to $tn^*$ (on top of $tn$). $\pi$ is executable in $s_0$; therefore, $tn^*$ is executable in $s_0$ because removing the propositions $e_i$ and the rule preconditions preserves the executability.

If the classical planner is complete, our algorithm is complete as it can always find $tn^*$ for a given $tn$, if it exists, and the algorithm generates all $tn$ in the non-decreasing length so it will eventually generate appropriate decomposition, if it exists. If the plan does not exist, the method may not terminate.

If the classical planner is optimal, then $tn^*$ is the shortest plan containing $tn$. We continue generating alternative plans $tn$ until we explore all that are shorter than the shortest plan $tn^*$ obtained. Therefore, our algorithm returns the shortest TIHTN plan, and hence it is optimal.

*3) Resolving preconditions on the fly:* For some problems, AND/OR trees can be quite large and different possibilities in OR-nodes may result in a large number of possible unambiguous decomposition trees. As a result, we might have to call a classical planner unsuccessfully for many similar primitive networks, which may differ in only a few actions. Therefore, we might want to call a classical planner on the fly for each discovered unsatisfied set of preconditions to rule out decomposition trees which cannot be extended to executable networks. Depending on a type of problem, this approach can be faster or slower than the previously described one.

To obtain an executable network from the corresponding AND/OR tree, we exhaustively traverse the tree using depth-first search. At each OR node, we check whether the preconditions of the corresponding rule are satisfied in the state

resulting from the execution of the already chosen actions. Let $a_1, ..., a_k$ be the actions corresponding to the leaves that have been already selected. As we traverse the leaves of the tree (actions) from the left to the right and the subtasks of each decomposition rule are totally ordered, we know that the first action into which the first subtask of the current rule will be decomposed will be executed in the state $s_k$ resulting from execution of the actions $a_1, ..., a_k$ in the initial state $s_0$. Similarly, when reaching a leaf of the tree, we check whether all the preconditions of the corresponding action are satisfied. If the preconditions of an action or a rule are not satisfied, we call a classical planner to find a classical plan to satisfy the preconditions.

Let us again define $R_a$ as the set of all rules whose first subtask decomposes into a sequence of actions starting by $a$ and, in a similar manner, $R_r$ be the set of all rules whose first subtask decomposes into the first subtask of rule $r$. Like $R_a$, $R_r$ can be obtained by following the edges of the AND/OR tree from the root to $r$. Once we reach a rule or an action whose preconditions are not satisfied, we will attempt to satisfy them while ensuring that the already satisfied preconditions of the previous rules remain satisfied.

The goal of the classical planner will be to satisfy the preconditions of the action $a$ or the rule $r$ from the state $s_k$. The planning problem can be defined as follows (notice that the goal has now positive and negative parts) :

$$P^C = P,$$
$$A^C = A^I,$$
$$s_0^C = s_k,$$

$$g^+ = \begin{cases} \text{prec}^+(a) \cup \bigcup_{r \in R_a} \text{prec}^+(r) & \text{for action } a, \\ \text{prec}^+(r) \cup \bigcup_{r' \in R_r} \text{prec}^+(r') & \text{for rule } r, \end{cases}$$

$$g^- = \begin{cases} \text{prec}^-(a) \cup \bigcup_{r \in R_a} \text{prec}^-(r) & \text{for action } a, \\ \text{prec}^-(r) \cup \bigcup_{r' \in R_r} \text{prec}^-(r') & \text{for rule } r. \end{cases}$$

Let $\pi$ be the plan found by the classical planner. We will append $\pi$ at the end of the action sequence $a_1, ..., a_k$ and resume the search in the decomposition tree.

If the classical planner fails to find the required plan, we return back in the tree towards the last OR node where another option can be selected. If we cannot create any executable network from the AND/OR tree, we resume the Earley parsing until another AND/OR tree is found.

This alternative approach still complies with our definition of a TIHTN planning problem since the inserted actions can be inserted non-deterministically into the found primitive network. The approach is also sound (assuming the soundness of the classical planner). However, in general, this approach is not complete. In some domain models, a plan found to satisfy preconditions of one action or rule may lead to unsatisfiability of preconditions of another action or rule later in the plan, e.g., some resources can be depleted in a plan to satisfy preconditions of action $a$, even though they are necessary to satisfy the preconditions of the following action $a'$ and they could be saved by executing a different plan for action $a$. If such dead ends are not possible in a domain model, i.e., if any planning state is reachable from any other planning state, this approach is complete. However, optimality still cannot be guaranteed, even if the classical planner is optimal. This is because despite each inserted classical plan is optimal, the complete plan may be longer – a suboptimal classical plan might be used for some part, which allows the classical plans for other parts to be shorter.

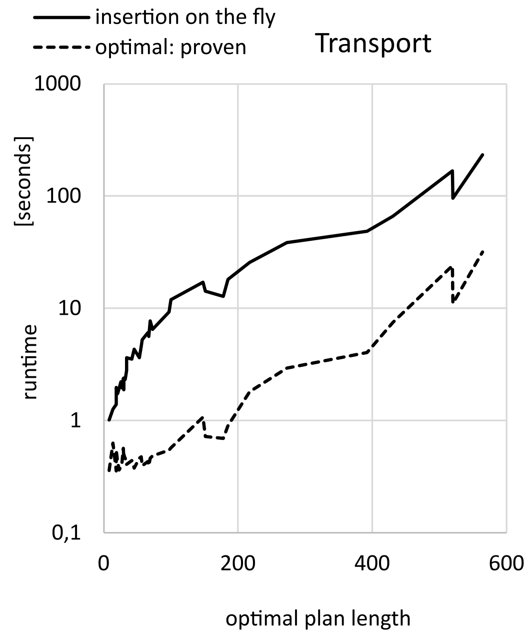


Fig. 2. Runtime in the domain Transport based on plan length (logarithmic y-axis).

## V. Empirical evaluation

We have evaluated the performance (runtime and quality of the solutions) of both variants of our algorithm. All experiments were performed on a computer with the Intel Core i7-11370H @ 3.30GHz processor and 16 GB of RAM with five minutes as the maximum allowed runtime for each TIHTN planning problem. The source codes and the benchmarks used for experiments are accessible on-line[1]. For classical planning, we used Fast Downward with A* search and the landmark-cut heuristic.

As there are no existing standard benchmark HTN domains that require TIHTN planning, we created two hierarchical domain models that use pathfinding, but their HTN models do not take pathfinding into account. The first domain was created by removing pathfinding from decomposition rules of

[1] https://github.com/krpant/TIHTN_Earley_planning

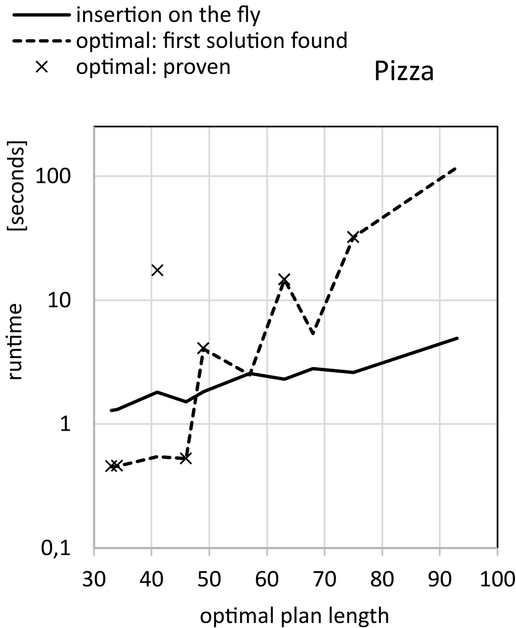


Fig. 3. Runtime in the domain Pizza (logarithmic y-axis).

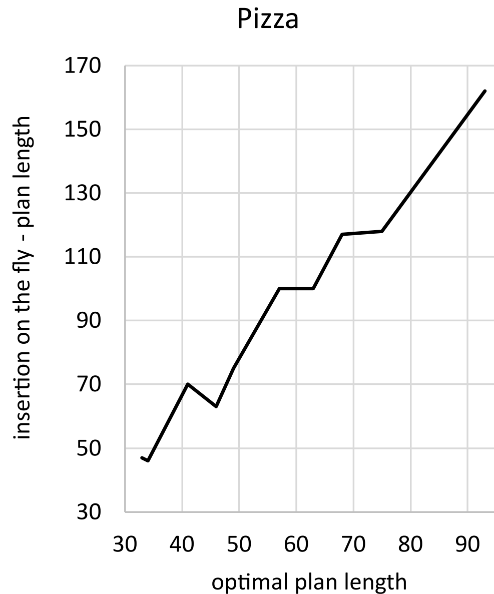


Fig. 4. Length of a found plan in the domain Pizza.

the totally ordered domain Transport from the International Planning Competition (IPC) 2020[2]. We have used 36 problems with optimal plans consisting of 8 - 564 actions. Plans in this domain correspond to sequences of actions performing loading and unloading packages at specific locations, interleaved by sequences of actions performing movement between locations. There is no difference in length of plans found by the two variants of our algorithm as optimal classical subplans are independent and contribute to the optimal plan. Figure 2 displays how the runtime depends on the length of the optimal plan, where the majority actions was related to pathfinding. The graph compares the two variants of our algorithm: *optimal: proven* denotes the optimal version, where the classical planner is called for the primitive network to resolve unsatisfied preconditions, *insertion on the fly* denotes the second version with multiple calls to the classical planner. In this domain, the optimal version was faster as calling the classical planner multiple times resulted in a worse runtime.

The second domain was designed with the purpose of showing how the lengths of the plans can differ for the two variants. The hierarchical model describes recipes for pizza but does not take into account the necessity to gather the ingredients for the recipe. We have created 10 problems with an optimal plan length from 34 to 93. Figure 4 compares the lengths of the plans produced by each algorithm for each problem. The differences are quite significant. Where the *optimal* approach managed to find an efficient plan in which more resources were often carried at the same time, *insertion on the fly* planned a separate path for each ingredient.

[2]https://github.com/panda-planner-dev/ipc2020-domains

However, Figure 3 shows that optimality comes at the expense of time. For the optimal variant, we also compare the runtime after which the solution was found and the time that was necessary to prove that the solution is optimal (by continuing with the search until it was clear that no better solution could be found). These two runtimes differed significantly for one problem and for the other three problems, the optimality proof had not been finished before the time limit passed.

In this domain, *insertion on the fly* was faster than the *optimal* approach in half of the problems. Although the runtimes of *insertion on the fly* remained similar for simpler and more difficult problems, the *optimal* version was significantly slower for the most difficult problems. In this domain, the decomposition rules had to be selected on the basis of the availability of ingredients, which had to be discovered by the classical planner. Therefore, the optimal planner called the classical planner for many primitive networks that could not be extended to executable networks. Furthermore, the optimal classical planner sometimes also required more time for a more difficult problem than for a set of simpler independent problems.

## VI. Conclusion

We proposed and implemented the parsing-based planner for totally ordered HTN planning problems with task insertion. The planner combines HTN planning based on the Earley parser with a classical planner to fill the gaps in the HTN plan. Two approaches were tried to insert actions. One that adds all the missing actions together at the end using the concept derived from plan recognition. This approach is complete and guarantees to find the shortest possible plan if a plan

exists. The second approach fills the gaps in the generated plan on the fly to satisfy missing preconditions of actions and rules the HTN planner used. This approach is neither optimal nor complete, but can be faster for some problems, where unsatisfiable preconditions might be discovered earlier and prune the search branch for the HTN planner. Extension to partially ordered domains is a topic of future work.

## References


[1] R. Alford, U. Kuter, and D. S. Nau. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In C. Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1629–1634, 2009.

[2] R. Alford, P. Bercher, and D. Aha. Tight bounds for HTN planning with task insertion. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1502–1508. IJCAI, 2015.

[3] R. Alford, G. Behnke, D. Höller, P. Bercher, S. Biundo, and D. W. Aha. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In A. J. Coles, A. Coles, S. Edelkamp, D. Magazzeni, and S. Sanner, editors, *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 20–28. AAAI Press, 2016.

[4] G. Behnke, F. Pollitt, D. Höller, P. Bercher, and R. Alford. Making translations to classical planning competitive with other HTN planners. In *Proceedings of 36th AAAI Conference on Artificial Intelligence*, pages 9687–9697. AAAI Press, 2022. doi: 10.1609/AAAI.V36I9.21203.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] K. Erol, J. A. Hendler, and D. S. Nau. Complexity results for HTN planning. *Annals of Mathematics and AI*, 18 (1):69–93, 1996.

[7] T. A. Estlin, S. A. Chien, and X. Wang. An argument for a hybrid HTN/operator-based approach to planning. In *Recent Advances in AI Planning: 4th European Conference on Planning (ECP)*, pages 182–194. Springer, 1997.

[8] T. Geier and P. Bercher. On the decidability of HTN planning with task insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1955–1961. IJCAI/AAAI, 2011.

[9] A. Gerevini, U. Kuter, D. Nau, A. Saetti, and N. Waisbrot. Combining domain-independent planning and HTN planning: The duet planner. In *Proceedings of 18th European Conference on Artificial Intelligence (ECAI)*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 573–577. IOS Press, 2008.

[10] M. Helmert. The Fast Downward planning system. *J. of Artificial Intelligence Research*, 26:191–246, 2006.

[11] D. Höller. Translating totally ordered HTN planning problems to classical planning problems using regular approximation of context-free languages. In S. Biundo, M. Do, R. Goldman, M. Katz, Q. Yang, and H. H. Zhuo, editors, *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS)*, pages 159–167. AAAI Press, 2021.

[12] L. P. Kaelbling and T. Lozano-Pérez. Hierarchical task and motion planning in the now. In *IEEE International Conference on Robotics and Automation*, pages 1470–1477, 2011. doi: 10.1109/ICRA.2011.5980391.

[13] S. Kambhampati, A. D. Mali, and B. Srivastava. Hybrid planning for partially hierarchical domains. In J. Mostow and C. Rich, editors, *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pages 882–888. AAAI Press / The MIT Press, 1998.

[14] J.-P. Kelly, A. Botea, and S. Koenig. Offline planning with hierarchical task networks in video games. In *The AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 60–65, Sep. 2021. doi: 10.1609/aiide.v4i1.18673.

[15] K. Pantůčková and R. Barták. Using Earley parser for recognizing totally ordered hierarchical plans. In *Proceedings of 26th European Conference on Artificial Intelligence (ECAI)*, pages 1819–1826. IOS Press, 2023.

[16] K. Pantůčková and R. Barták. Correcting totally ordered hierarchical plans by action deletion and insertion. *Proceedings of the 7th ICAPS Workshop on Hierarchical Planning (HPlan 2024)*, page 27–35, 2024.

[17] K. Pantůčková, S. Ondrčková, and R. Barták. Using Earley parser for verification of totally ordered hierarchical plans. *Proceedings of The International FLAIRS Conference*, 37(1), 2024.

[18] M. Ramírez and H. Geffner. Plan recognition as planning. In C. Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1778–1783. IJCAI, 2009.

[19] B. Schattenberg and S. Biundo. A unifying framework for hybrid planning and scheduling. In *Advances in Artificial Intelligence: 29th Annual German Conference on AI (KI)*, pages 361–373. Springer, 2006.

[20] V. Shivashankar, R. Alford, U. Kuter, and D. S. Nau. The GoDeL planning system: A more perfect union of domain-independent and hierarchical planning. In F. Rossi, editor, *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2380–2386. IJCAI/AAAI, 2013.

[21] Z. Xiao, H. Wan, H. H. Zhuo, A. Herzig, L. Perrussel, and P. Chen. Refining HTN methods via task insertion with preferences. In *Proceedings of The 34th AAAI Conference on Artificial Intelligence*, pages 10009–10016. AAAI Press, 2020. doi: 10.1609/AAAI.V34I06.6557.