Correcting Totally Ordered Hierarchical Plans by Action Deletion and Insertion

Kristýna Pantůčková, Roman Barták

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
{pantuckova, bartak}@ktiml.mff.cuni.cz

Abstract

Hierarchical planning extends classical planning by capturing the hierarchical structure of tasks. Plan correction, an extension of plan verification, involves assessing the validity of a given plan. When an input plan is invalid, the solver searches for a valid alternative plan closest to the original plan. Currently, the only approach for plan correction solely supports action deletion from the original plan. This paper presents the first approach supporting action insertion and deletion within totally ordered hierarchical domain models. Moreover, this new approach is more efficient than the existing technique when only action deletion is allowed.

Introduction

Hierarchical planning (Erol, Hendler, and Nau 1996) extends classical planning by modeling the natural hierarchy of tasks. Complex tasks decompose into simpler subtasks until the simple executable tasks (actions) are reached. The goal of a hierarchical planner is to find such a decomposition for a given root task. Conversely, plan verification aims to determine whether a given sequence of actions is a valid decomposition of any task. Plan correction extends plan verification by providing a valid plan that can be obtained from the invalid plan by a minimum number of corrections.

In real-world scenarios, observations of plans are frequently incomplete or even incorrect. Such observations may encompass actions that have not been executed, or the observer might overlook some actions, leading to an incomplete plan. In this case, a mere statement that the observed plan is invalid may not be the desired answer. Plan correction techniques provide a valid alternative plan that is as close to the observed sequence of actions as possible.

The pioneering work introducing hierarchical plan correction (Barták et al. 2021) defined two possible correction steps: action deletion and action insertion. However, the correction technique presented there supported action deletion only. This paper proposes a novel approach to correct totally ordered hierarchical plans by action deletion and insertion. In totally ordered hierarchical domain models, complex tasks decompose into totally ordered sequences of subtasks. Totally ordered domains can naturally describe many problems. For example, the International Planning Competition (IPC) 2020 used 33 hierarchical domain models, of which 24 were totally ordered.

This paper presents the first approach to correcting totally ordered hierarchical plans supporting both action deletion and insertion. Moreover, if the proposed solver corrects plans solely by action deletion, it is faster than the only existing approach (Barták et al. 2021). HTN plan correction by action deletion and action insertion leads to many possible applications including:

- HTN plan verification (when neither action deletion nor action insertion is enabled), see (Pantůčková, Ondrčková, and Barták 2024);
- correcting an HTN plan for a known goal (when the possible top-level task is given);
- HTN plan recognition with full observability (when action deletion is disabled and action insertion is enabled only after the observed plan prefix), see (Pantůčková and Barták 2023);
- HTN plan recognition with partial observability (when action deletion is disabled and action insertion is enabled anywhere in the observed sequence);
- HTN plan recognition with full or partial observability and with noise (when action deletion is enabled);
- HTN planning (when action insertion is enabled and the input plan is empty).

The paper is organized as follows. We first provide the necessary background on hierarchical plan correction and summarize the related work. Then, we describe the novel plan correction algorithm based on Earley parser, and finally, we present empirical evaluation results. We compare the performance of plan correction by action deletion with the existing technique (Barták et al. 2021). Then, we study the performance of action deletion and action insertion separately, and finally, we assess the performance of our solver when both means of plan correction are allowed.

Background on hierarchical plan correction

Hierarchical planning focuses on planning problems where complex (abstract) tasks decompose into simpler subtasks until a sequence of indecomposable tasks (actions) is reached. Similarly to classical planning, actions are defined by preconditions (propositions that must hold in the state where the action will be executed) and effects (propositions

that will hold after the action is executed). Hierarchical planning is often described by the formalism of hierarchical task networks (HTN).

A domain model can be defined as $\mathcal{D} = (P, T, A, R)$, where P is a set of predicates, T is a set of abstract tasks, A is a set of actions and R is a set of decomposition rules. A rule $T \rightarrow T_1, ..., T_n\ [C]$ decomposes the task T into subtasks $T_1, ..., T_n$. C is a set of rule constraints, which can contain ordering conditions, before-constraints, and between-constraints:

- an ordering constraint $T_i \prec T_j$ enforces the order of actions into which the tasks T_i and T_j will be decomposed; i.e., the last action of the task T_i must be executed before the first action of the task T_j;
- *before*(T', p), indicates that the proposition p must hold in the state in which the first action of the first task in the set T' is executed; and
- *between*(T', T'', p) indicates that p must hold in all states between the execution of the last action of the tasks in T' and the execution of the first action of the tasks in T''.

In a totally ordered domain model, actions into which the subtasks decompose must be executed in the given order, i.e., all actions of T_i are executed before the actions of T_{i+1}.

Given an observed sequence of actions, plan correction aims to find a task that decomposes into a sequence of actions closest to the observed sequence, where the number of corrections measures the distance between plans. We allow corrections of two types: deleting one of the observed actions or inserting a new action into the observed plan (Barták et al. 2021).

Formally, a *plan correction problem* is defined as $\mathcal{P} = (\mathcal{D}, C, I, O)$, where $\mathcal{D}$ is a domain model, C is a set of constants, $O =< o_1, ..., o_n >$ is a sequence of observed actions and I is the initial state (a set of propositions that were valid before the actions were executed). A (optimal) solution to the plan correction problem is an action sequence $\pi =< a_1, ..., a_m >$ such that:

- π is a valid plan with respect to the domain model, that is, π is executable at state I and there exists some task that decomposes to π with respect to domain model $\mathcal{D}$,
- there is a function $f : \{1, ..., n\} \rightarrow \{1, ..., m\} \cup \{nil\}$ such that
  - $\forall i < j : f(i) < f(j) \vee f(i) = nil \vee f(j) = nil$ (actions are not swapping positions),
  - $\forall i : f(i) \neq nil \implies o_i = a_{f(i)}$ (actions preserved in the plan),
  - $del = \{i | f(i) = nil\}$ (actions deleted from O),
  - $add = \{j | \neg\exists i : f(i) = j\}$ (actions added to π),
  - $|del| + |add|$ is minimal among all plans π.

Example 1. *Figure 1 contains an example of a hierarchical task network where the goal is to deliver the package pkg1 to the location loc3. The root task deliver(pkg1, loc3) decomposes into three subtasks: load the package at the location loc1 (pickup(pkg1, loc1)), go to the target location loc3 (get_to(loc3)) and unload the package at the location loc3 (drop(pkg1, loc3)). The middle subtask get_to(loc3) is realized by driving from the location loc1 to loc3 through the location loc2, therefore, it decomposes into two subtasks: drive(loc1, loc2), drive(loc2, loc3). The leaf tasks (pickup(pkg1, loc1), drive(loc1, loc2), drive(loc2, loc3), and drop(pkg1, loc3)) are actions, which can be executed, the internal nodes (deliver(pkg1, loc3) and get_to(loc3)) are abstract tasks.*

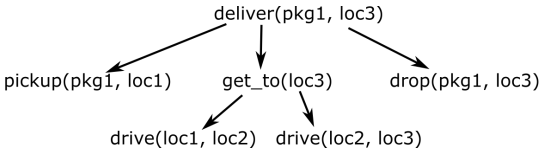

Figure 1: An example of a hierarchical task network.

The decomposition tree includes two decomposition rules: one rule decomposes the top-level task deliver(pkg1, loc3) into three subtasks, and the other rule decomposes the middle subtask get_to(loc3) into two subtasks. Subtasks of both rules are totally ordered. The preconditions of the action pickup(pkg1, loc1) ensure that both the package pkg1 and the truck are at the location loc1 and its effect states that pkg1 is loaded into the truck. Similarly, preconditions of drop(pkg1, loc3) ensure that the truck is at loc3, and after this action is executed, pkg1 will no longer be loaded into the truck. Each action drive must be executed in the first location so the truck will move to the second location.

Related works

The problem of deciding whether a given HTN plan is valid is tackled by plan verification techniques. Existing approaches are based on parsing (Lin et al. 2023), compilation to SAT (Behnke, Höller, and Biundo 2017), or compilation of HTN plan verification problems to planning problems (Höller et al. 2022). Another related problem is HTN plan recognition, where a prefix of an HTN plan is given as part of the input, and the goal is to find an abstract task that can be decomposed into a plan with the given prefix. Similarly to plan verification, recent existing approaches are based either on parsing (Barták, Maillard, and Cardoso 2020) or on compilation to HTN planning (Höller et al. 2018). The Earley parser used in this paper was first proposed for HTN plan recognition by Pantůčková and Barták (2023).

The only other existing approach to HTN plan correction (Barták et al. 2021) corrects plans by action deletion. This parsing-based approach greedily composes from the bottom upward all abstract tasks that can be composed of the available actions using the decomposition rules from the domain model while deleting actions from the input plan that violate preconditions of the actions used in the composition. The algorithm exhaustively performs all such compositions until no new tasks can be composed. In contrast to the approach from this paper, the existing solver (Barták et al. 2021) can be used on partially ordered domain models. On the other hand, our approach supports both action insertion and action deletion; therefore, it can also be used for plans with missing (unobserved) actions in addition to plans with noisy observations. Our solver also demonstrates superior performance

if only action deletion is allowed, as top-down parsing in our solver leads to fewer intermediate solutions than greedy bottom-up parsing.

Another weakly related problem is plan repair (Zaidins, Roberts, and Nau 2023), (Goldman, Kuter, and Freedman 2020), (Höller et al. 2020), which aims to make changes in a not-yet executed suffix of a previously created plan at a point of execution when conditions change and it is not possible to continue with execution of the plan. This problem differs from plan correction, which modifies an action sequence to obtain a valid hierarchical plan.

Aho and Peterson (1972) proposed an approach to find a minimum edit distance string in context-free grammars, which is based on a similar idea as our approach. They extend the given grammar by rewriting rules representing all types of corrections and then they use the Earley parser straightforwardly. However, the search progresses differently. The approach of Aho and Peterson (1972) fills the table of the original Earley parser (i.e., finds all possibilities how the input sequence can be corrected). Then, the decomposition with the fewest corrections is found in the table. In contrast, our search is based on a priority queue, which takes into account the number of corrections required in each intermediate solution. Therefore, the search can terminate earlier – when there are no states which could lead to a better solution than the one that has already been found. Moreover, our approach does not insert new rewriting rules into a grammar. In contrast to the work the work of Aho and Peterson (1972), we generate correction rules one by one when they are needed. While Aho and Peterson (1972) works exclusively with context-free grammars, we include also evaluation of constraints. Furthermore, our approach is lifted (tasks have attributes).

Correction by action deletion and insertion

We present a lifted HTN plan correction approach based on top-down parsing. By omitting constraints of decomposition rules in a totally ordered domain model, we obtain an abstraction to a context-free grammar (CFG). CFG is a formal grammar whose rewriting rules are $A \rightarrow \alpha$, where A is a non-terminal symbol and α is a sequence of symbols (terminal and non-terminal). We propose a totally ordered HTN plan correction algorithm inspired by the Earley parser (Earley 1970), a top-down CFG parser. In contrast to (Barták et al. 2021), where parsing progresses from the bottom (from the available actions) upwards, the Earley parser progresses from the possible top-level tasks downwards to actions. Top-down parsing provides a considerable advantage for action insertion as it allows one to determine which actions must be generated to decompose candidate goals.

Earley parser starts by decomposing all possible top-level tasks into subtasks and then decomposing these abstract tasks until the actions are reached. The parser is based on dynamic programming. The input sequence is parsed from the left to the right as the parser systematically processes states by the index of the last symbol covered in the input sequence. The systematic left-to-right approach is unsuitable for optimal plan correction since we need to process states based on the number of corrections and not the end index. Therefore, we use a priority queue instead of a table. In addition to the priority queue, we also need to remember a set of all states that have been generated. For each state, we define its cost as the minimum number of corrections that will be done in the plan if the rule is used. The priories of enqueued states correspond to their costs, i.e., states from the queue are dequeued based on their cost (states with the lowest cost are dequeued first).

In the plan correction setting, the actions can be selected from the input plan or inserted into the plan. Successful lower-level decompositions can be used to complete higher-level decompositions until a decomposition of a top-level task is found.

The priority queue contains states of the following form:

$$s = (T_1 \rightarrow T_2 ... T_r \bullet T_s ... T_t, i, j).$$

This state represents a decomposition rule, which decomposes task T_1 into subtasks $T_2, ..., T_t$. Symbol $\bullet$ separates subtasks that have been already successfully decomposed (i.e., some lower-level rules decomposing $T_2, ..., T_r$ have been completely decomposed into actions) from subtasks that still remain to be decomposed. Each state is supposed to represent a possible coverage of a continuous subsequence of actions from the input plan; i is the index of the last action covered before this state (i.e., i indicates where in the input plan the decomposition of the first subtask of this rule should start) and j is the index of the last action covered by this state so far (i.e., the last action into which the last task before $\bullet$ decomposes).

The plan correction algorithm starts by enqueueing a state

$$(I \rightarrow \bullet T, 0, 0)$$

for each abstract task T, where I is a dummy starting task. Therefore, we do not need to know the goal task for which the input plan was generated as we take into account all possible goals. All these states have cost equal to zero, corresponding to zero corrections introduced so far.

The Earley parser defines three procedures for processing states of three different types: *completer*, *predictor* and *scanner*. **Scanner** processes states, where the first task, that has not been decomposed yet, is an action. For a state

$$s = (T_1 \rightarrow T_2 ... \bullet T_m ..., i, j),$$

where T_m is an action, we can create multiple new states. The corresponding action can be either taken from the input plan, where some actions from the plan may have to be skipped, or the action is inserted into the plan. If there is action a_{j+1} in the input plan unifiable with with (partially) instantiated primitive task T_m, we create a new state (with $\bullet$ shifted after T_m)

$$s' = (T_1 \rightarrow T_2 ... T_m \bullet ..., i, j+1).$$

The state s' will be enqueued into the priority queue with the priority equal to $priority(s)$.

The possible corrections that can be generated by a scanner state $(T_1 \rightarrow T_2 ... \bullet T_m ..., i, j)$ are the following (k can also be zero):

- delete k actions $(a_{j+1}, ..., a_{j+k})$ from the input sequence and unify action a_{j+k+1} with T_m with $cost = k$;

- delete k actions $(a_{j+1}, ..., a_{j+k})$ from the input sequence and insert a new action T_m after the deleted actions (between a_{j+k} and a_{j+k+1}, if a_{j+k} was not the last action in the input sequence) with $cost = k + 1$.

A scanner state will generate one correction at a time and enqueue itself into the priority queue again.

Let d be the number of deleted actions. If the action for T_m was selected from the plan, we will generate a new state

$$s'' = (T_1 \rightarrow T_2 ... T_m \bullet ..., i, j + d + 1),$$

whose priority will be equal to $priority(s)+d$ (d corrections introduced). If the new action had to be inserted into the plan, the priority of the new state

$$s''' = (T_1 \rightarrow T_2 ... T_m \bullet ..., i, j + d)$$

will be equal to $priority(s) + d + 1$.

Our approach does not require a grounded domain as an input. At the beginning, the priority queue is filled with the artificial initial starting rules for all possible uninstantiated abstract tasks. When scanner selects an action from the input plan, its variables will be propagated upwards into the newly created states. When a new action is inserted into the plan, it will be (partially) instantiated with variables propagated downwards from partially instantiated rules (by predictor).

When a scanner state is dequeued, we perform the next correction (the correction with the lowest priority) and enqueue the scanner state back into the queue with an increased priority and with the information which correction should be performed next.

Predictor is used to process states where the first task that has not been decomposed yet is an abstract task. For a state

$$s = (T_1 \rightarrow T_2 ... \bullet T_m ..., i, j),$$

where T_m is an abstract task, we create a state

$$s' = (T_m \rightarrow \bullet T_o^m ... T_p^m, j, j)$$

for each decomposition rule decomposing the task T_m. As the priority of states in the queue should be equal to the minimum number of corrections, we will enqueue s' with the priority equal to the priority of s as if the state s' is used in a decomposition, the state s must also have been used before. If the same state as s' is generated later by another state s'' with a priority lower than s, the priority of s' will be updated to the priority of s''. If the state s' already exists, we do not enqueue it again; we only update its priority.

Completed states (states, where all subtasks have been decomposed) are processed by **completer**. A state

$$s = (T_1 \rightarrow T_2 ... T_m \bullet, i, j)$$

represents a possible decomposition of the task T_1 covering actions $a_{i+1}, ..., a_j$. This state can be used to decompose the task T_1 whose decomposition should start by a_{i+1}. Therefore, for each state (containing T_1 right after $\bullet$)

$$s' = (T_0 \rightarrow ... \bullet T_1 ..., k, i)$$

we create a new state (with shifted $\bullet$)

$$s'' = (T_0 \rightarrow ... T_1 \bullet ..., k, j).$$

The number of corrections in s'' will be equal to the number of corrections in s' + the number of corrections in s. The priority of s'' will be equal to its number of corrections + the minimum priority of the predictor states that generated the decomposition of T_0. If the state s'' already exists, we will remember the new relation between the completing and completed state (this information will be used later to build a decomposition tree for a candidate solution) and recompute its priority. Instead of adding the number of corrections in s, we will use the minimum number of corrections of all states providing a complete decomposition of T_1.

Each state

$$(I \rightarrow T \bullet, 0, j),$$

covering a prefix of the plan up to the index j represents a candidate top-level task (actions starting with a_j, if any, are supposed to be deleted from the input plan). When the queue does not contain a state with a priority lower than the priority of the best solution found so far, the algorithm stops as no better candidate top-level task can be found. The algorithm may be terminated earlier and return the best candidate top-level task found so far. Hence, the algorithm can be seen as an anytime technique.

During search, completer builds an AND/OR tree for each candidate top-level task, where OR-nodes are abstract tasks, which can be decomposed by multiple decomposition rules, AND-nodes are decomposition rules, which decompose the task into the given subtasks, and leaves are actions. A completer links a decomposition rule to an abstract subtask in another decomposition rule. To extract a solution from an AND/OR tree, it needs to be checked whether we can choose one decomposition rule in each OR-node and ground all ungrounded variables in all nodes of the tree such that preconditions of all actions and all constraints of all decomposition rules are satisfied (with respect to the initial state and to state transitions after each action). We traverse AND-OR trees in a DFS-like manner.

Example 2. *Consider the HTN from Figure 1, let the initial state define all roads as road(loc1, loc2), road(loc2, loc3) and road(loc4, loc3) and let pickup(pkg1, loc1), drive(loc1, loc2), drive(loc4, loc3), drop(pkg1, loc3) be the invalid input plan. At the beginning, our algorithm will enqueue two states into an empty priority queue (assuming that there are only two abstract tasks in the domain):*

$$[(I \rightarrow \bullet deliver(?, ?), 0, 0), priority = 0] \quad (1)$$
$$[(I \rightarrow \bullet get\_to(?), 0, 0), priority = 0].$$

When state (1) is dequeued from the queue, predictor will create one new state:

$$[(deliver(?, ?) \rightarrow \bullet pickup(?, ?), get\_to(?), drop(?, ?), 0, 0), priority = 0]. \quad (2)$$

State (2) will be processed by scanner. As the first action in the input plan can be used as the desired pickup action, a new state can be created without any corrections:

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), \bullet get\_to(?), drop(pkg1, ?), 0, 1), priority = 0]. \quad (3)$$

For state (3), predictor will create a new state based on available decomposition rules. The decomposition of the new state should start by the second action in the input plan, as the first action is already covered:

$$[(get\_to(?) \rightarrow \bullet drive(?,?), drive(?,?), 1, 1), priority = 0]. \quad (4)$$

Again, scanner can use the second action in the plan to cover the first subtask in state (4) and create the next state:

$$[(get\_to(?) \rightarrow drive(loc1, loc2), \bullet drive(loc2, ?), 1, 2), priority = 0]. \quad (5)$$

However, the next action in the plan drive(loc4, loc3) is not unifiable with drive(loc2, ?). The desired action can be inserted into the plan and so the scanner will insert drive(loc2, ?) before drive(loc4, loc3). Since the resulting state required 1 correction, its priority value will be higher than the priority of state (5):

$$[(get\_to(?) \rightarrow drive(loc1, loc2), drive(loc2, ?)\bullet, 1, 2), priority = 1]. \quad (6)$$

State (6) will be processed by completer and used to complete the next subtask of state (3):

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), get\_to(?), \bullet drop(pkg1, ?), 0, 2), priority = 1]. \quad (7)$$

The next action of state (7) is again not unifiable with the next action in the input plan (drive(loc4, loc3)). Scanner will firstly insert the required action drop(pkg1, ?) into the input plan before drive(loc4, loc3), thus increasing the number of corrections, and enqueue the new state:

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), get\_to(?), drop(pkg1, ?)\bullet, 0, 2), priority = 2]. \quad (8)$$

As another possible correction, the scanner can skip the action drive(loc4, loc3) and select the next action from the plan. Therefore, it will enqueue again the state 7 with priority equal to 2:

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), get\_to(?), \bullet drop(pkg1, ?), 0, 3), priority = 2]. \quad (9)$$

Completer can then use the completed state (8) to complete the state 1 and create a candidate top-level rule:

$$[(I \rightarrow deliver(pkg1, ?)\bullet, 0, 2), priority = 2]. \quad (10)$$

We will then attempt to extract a solution from state (10). The missing variable (the location to which the package pkg1 will be delivered) can be chosen arbitrarily from the set of the locations to which the truck can drive from loc2. However, the resulting plan will require two more corrections (four corrections in total) as the last two actions from the input plan will be deleted. As there is still a state with fewer than four corrections (state (9)), the algorithm continues to find a better correction.

Scanner will then process state (9) and create a new state, where the action drive(loc4, loc3) will be deleted from the plan and the last action in the plan will be covered:

$$[(deliver(pkg1, ?) \rightarrow pickup(pkg1, loc1), get\_to(loc3), drop(pkg1, loc3)\bullet, 0, 4), priority = 2]. \quad (11)$$

State (11) will be processed by completer and another candidate top-level rule will be created:

$$[(I \rightarrow deliver(pkg1, loc3)\bullet, 0, 4), priority = 2]. \quad (12)$$

State (12) can be used to extract the desired valid plan, which can be created from the input plan by two corrections: deleting the third action from the plan and inserting a different action into its position.

Theorem 1. *The plan correction algorithm is sound.*

Sketch of proof. The correctness is implied by the soundness of the Earley algorithm. The modified Earley algorithm builds AND/OR trees representing all possible decompositions and then the algorithm selects in the internal nodes the decomposition rules whose constraints are satisfied and in the leaves actions which are executable with respect to their preconditions and effects. □

Theorem 2. *The plan correction algorithm is complete.*

Sketch of proof. We will first prove that the algorithm terminates. Each *scanner* state can only generate a finite number of descendant states (with nondecreasing costs), where the last state is created by the insertion of the requested action after the sequence of observed actions (skipping all preceding actions).

As *completer* and *predictor* procedures work similarly as in the original Earley parser, finiteness of the Earley parser implies finiteness of the HTN plan correction algorithm. Let us note that finiteness is guaranteed even for models with recursive decomposition rules as predictor does not create and enqueue states that already exist. If the requested descendant state ($T \rightarrow \bullet$..., with the suitable starting index) has already been created by another predictor, predictor will not generate the state again, preventing infinite recursion. This is one of the basic principles of the original Earley parser. E.g., if there are two rules $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$, the predictor state $s = T_1 \rightarrow \bullet T_2$ will generate a new state $s' = T_2 \rightarrow \bullet T_1$, but when s' is processed, s will not be generated again.

Therefore, if there does not exist any plan that can be generated by the rules available in the domain model, the plan correction algorithm will terminate when the queue is exhausted. If a valid plan exists, the algorithm is guaranteed to find it (which is implied by the completeness of the original Earley parser), in the worst case by deleting all input actions and inserting new ones. In this case, the shortest possible plan will be found as the algorithm terminates when a plan with a lower cost cannot be found.

□

Theorem 3. *The plan correction algorithm is optimal.*

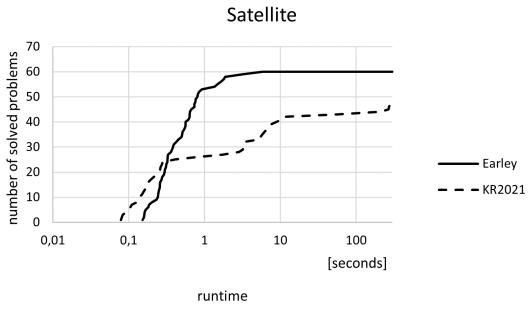

Figure 2: The number of problems solved within given time (logarithmic x-axis) in the domain Satellite. *KR2021* is the solver from (Barták et al. 2021), *Earley* is our solver.

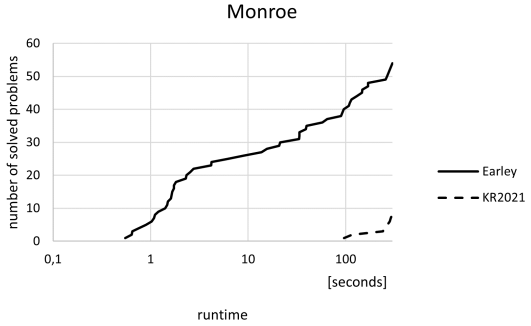

Figure 3: The number of problems solved within given time (logarithmic x-axis) in the domain Monroe. *KR2021* is the solver from (Barták et al. 2021), *Earley* is our solver.

Sketch of proof. For each state, we know the minimum number of flaws of a solution that can be extracted from this state. The algorithm halts and returns a plan only if there remain no untried top-level states which could yield a plan with a lower cost and if there are no states with a lower cost in the queue. As a state with cost c can only contribute to solutions with a cost $c' \geq c$, the solution is optimal. □

Empirical evaluation

The experiments were executed on a computer with the Intel Core i7-8550U CPU @ 1.80GHz processor and 16 GB of RAM. Maximum allowed runtime was set to five minutes for one problem. For the experiments, we assumed that the root task is not known, which is the same setting which was used for the plan correction by action deletion approach (Barták et al. 2021). This setting corresponds to HTN plan recognition with partial observability and noise.

We used domains and plans from the International Planning Competition (IPC) 2020. The valid plans consisted of 9 – 28 actions in the domain Satellite, 18 – 71 in Transport, 15 – 68 in Monroe and 22 – 168 in Blocksworld. For the task of correcting plans by action deletion, we added noise to the plans by inserting extra actions into valid plans. For plan correction by action insertion, we deleted some actions from

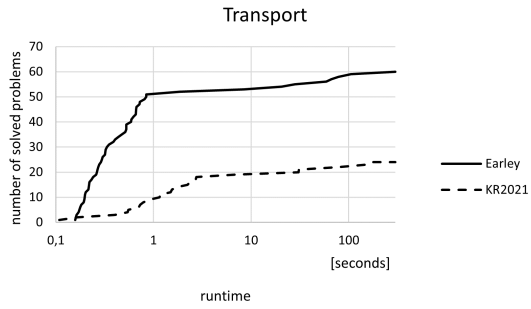

Figure 4: The number of problems solved within given time (logarithmic x-axis) in the domain Transport. *KR2021* is the solver from (Barták et al. 2021), *Earley* is our solver.

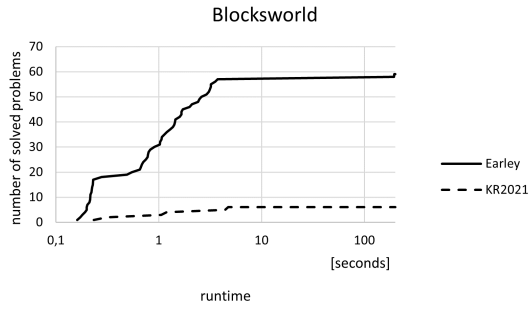

Figure 5: The number of problems solved within given time (logarithmic x-axis) in the domain Blocksworld. *KR2021* is the solver from (Barták et al. 2021), *Earley* is our solver.

valid plans. Let us note that the solution does not have to be the original valid plan as there may be a different valid plan which can be obtained from the modified plan by a lower or equal number of corrections. The resulting plans used for experiments are accessible on-line[1].

Correcting plans by action deletion

We have compared the performance of our approach with the approach of Barták et al. (2021). As this approach supports only action deletion, we have compared it with our approach also restricted to action deletion.

We have run both solvers on valid plans and on invalid plans, which were created from valid plans by inserting at least one and at most five extra actions. Adding more noise to the plans does not seem beneficial as it would lead the solvers to longer valid plans that could be achieved by deleting fewer actions. In the domain Blocksworld, the solvers already found valid plans which were longer than the original plan by one or two actions.

Figures 2, 4, 3 and 5 show how the total number of problems grows with runtime. Each figure shows results from a different domain. Each domain contains 60 plans: 10 different valid plans and 50 plans created by inserting extra ac-

[1] https://github.com/krpant/Plan-correction-benchmarks

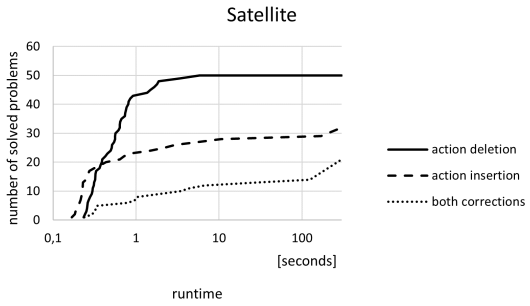

Figure 6: The number of problems solved within given time (logarithmic x-axis) from the domain Satellite using action deletion or insertion or by both means of correction.

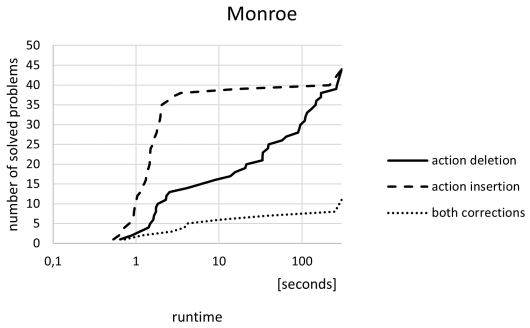

Figure 7: The number of problems solved within given time (logarithmic x-axis) from the domain Monroe using action deletion or insertion or by both means of correction.

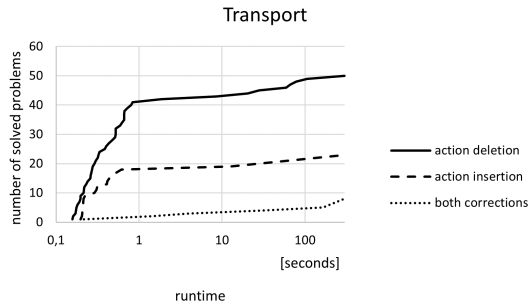

Figure 8: The number of problems solved within given time (logarithmic x-axis) from the domain Transport using action deletion or insertion or by both means of correction.

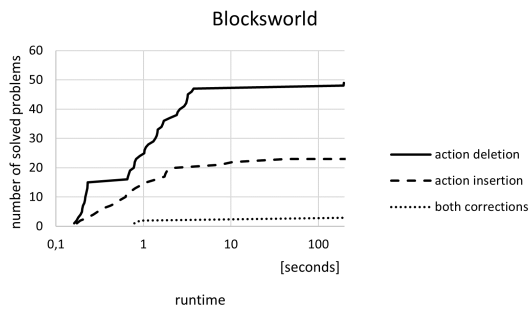

Figure 9: The number of problems solved within given time (logarithmic x-axis) from the domain Blocksworld using action deletion or insertion or by both means of correction.

tions into the valid plans. The graphs consider the runtime after which the computation stopped and provided the correct plan. Solver runs which found the correct solution but did not terminate before the time limit has passed are displayed with runtime equal to the time limit. Our approach outperformed the approach of Barták et al. (2021) on all four domains.

Correcting plans by action deletion and insertion

Action deletion vs action insertion Additional experiments were performed to assess how demanding is action insertion in comparison to action deletion. We have created another 50 invalid plans for each domain by deleting at least one and at most five actions from valid plans and we have measured how fast our solver corrected these plans solely by action insertion.

domain	ins	ins*	del	del*
Satellite	32	19	50	50
Transport	23	19	50	49
Monroe	44	40	44	40
Blocksworld	23	23	47	47
total	122	101	193	188

Table 1: Number of solutions found within 5 minutes by action insertion (*ins*) and deletion (*del*) and number of solutions where the optimality was proven(*ins** and *del**).

Figures 6, 8, 7 and 9 show how fast our solver solves action insertion problems in comparison to action deletion problems and problems of plan correction by action insertion and deletion simultaneously. Table 1 compares the number of problems solved within the given time limit solely by action insertion or deletion, considering separately all solutions with the expected number of corrections which were found within the time limit, and solutions where also the proof of optimality was found (i.e., the solver terminated before the time limit has passed). In general, action insertion seems to be more demanding than action deletion. In the domains Satellite, Transport and Blocksworld, the solver corrected significantly less plans by action insertion than by action deletion within the given time limit.

On the contrary, in the domain Monroe the solver eventually arrived to the same result for both types of problems and most action insertion problems were even solved faster than the action deletion problems. The domain Monroe defines a complex hierarchy of object types and a variety of specialized decomposition rules, while the plans from the other three domains consist of sequences of rather simple tasks

which can be composed using many different combinations of objects. Therefore, we assume that in these three domains it was more difficult to find the missing actions as there were too many possible actions that could be generated in order to complete the rules of the parser. On the other hand, plan correction with only action deletion limits completed rules to those that can be completed by existing actions.

In the domain Monroe, however, the number of possible decompositions of a rule was significantly lower; therefore, the solver was guided by the domain model to insert only the most relevant actions. Action deletion could then require more time simply because the input plan was longer, which resulted in more completed subtrees which could not be used to complete the decomposition of a top-level task.

Figure 10 shows how the number of problems from the domain Transport solved by different numbers of corrections grows with runtime. The figure consider the runtime after which the computation stopped and provided the correct plan. Solver runs, which found the correct solution but did not terminate before the time limit has passed, are displayed with runtime equal to the time limit. As expected, plans requiring more corrections were usually more difficult to correct. The difference was more noticeable on action insertion.

Both corrections simultaneously To assess the performance of the solver when both means of correction are enabled simultaneously, we have created more invalid input plans by deleting one or two actions and inserting one or two actions into valid plans; therefore, we have generated four invalid plans for each valid plan. However, for some plans it was possible to find a valid plan by less corrections than intended. Figures 6, 8, 7 and 9 show how the number of problems solved solely by action deletion or insertion or by both means of correction grows with runtime. The graphs consider the runtime after which the computation stopped and provided the correct plan. Solver runs which found the expected solution but did not terminate before the time limit has passed are displayed with runtime equal to the time limit. Unsurprisingly, the solver runs slower as more rules are created by the scanner. The solver was most successful in the domain Satellite, which contains in general the shortest plans. In this domain, the number of solved problems was close to the number of problems solved solely by action insertion. In the other domains, the solver corrected significantly less plans than by only one mean of correction. Even in the domain Monroe, where the complexity of both means of correction seems to be similar, the performance decreased considerably when both means of correction were enabled.

Solution quality over time Our algorithm incrementally improves the quality of its result in order to be able to provide the best possible result any time it is terminated. Therefore, it attempts to extract a solution from each candidate top-level rule whenever such a rule is found that could provide a solution with fewer corrections than the best solution found so far. Often the optimal solution was found early, though the algorithm required much more time to dismiss candidate partial solutions with less corrections and therefore prove that the solution is optimal (see the difference

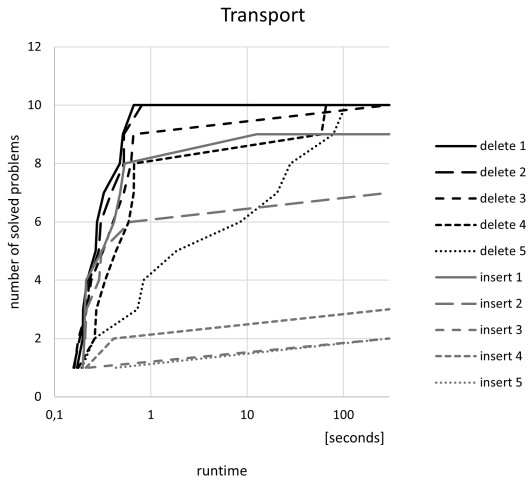

Figure 10: The number of problems solved withing given time (logarithmic x-axis) from the domain Transport using different means and numbers of corrections.

of solved problems with and without proof of optimality in Table 1). When only action insertion was allowed, the algorithm found for all problems in all domains only one complete solution. When action deletion was enabled, a sequence of solutions with improving quality was often found as more possible top-level tasks could be sometimes found simply by covering a prefix of the input plan and deleting the rest of the actions.

Conclusion

We propose a novel approach to correcting totally ordered HTN plans, the first approach facilitating both action deletion and action insertion. Furthermore, our approach outperforms the existing HTN plan correction approach when only action deletion is enabled. Future work may aim to enhance the efficiency of the solver when both means of correction are enabled, as the performance is better when only action deletion or insertion is allowed. Another possible research direction could focus on extending the algorithm to partially ordered HTN plans.

Acknowledgements

Research is supported by the Charles University, project GA UK number 156121, by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215 and by SVV project number 260 698.

References

Aho, A. V.; and Peterson, T. G. 1972. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4): 305–312.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2020. Parsing-based Approaches for Verification and Recognition of Hierarchical Plans. In *The AAAI 2020 Workshop on Plan, Activity, and Intent Recognition*.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021. Correcting hierarchical plans by action deletion. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 18, 99–109.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (... but is it though?) - verifying solutions of hierarchical planning problems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 27.

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

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Annals of Mathematics and AI*, 18(1): 69–93.

Goldman, R. P.; Kuter, U.; and Freedman, R. G. 2020. Stable plan repair for state-space HTN planning. In *Proceedings of the 3rd ICAPS Workshop on Hierarchical Planning (HPlan 2020)*, 27–35.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018. Plan and goal recognition as HTN planning. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence*, 466–473.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020. HTN plan repair via model transformation. In *KI 2020: Advances in Artificial Intelligence: 43rd German Conference on AI, Bamberg, Germany, September 21–25, 2020, Proceedings 43*, 88–101. Springer.

Höller, D.; Wichlacz, J.; Bercher, P.; and Behnke, G. 2022. Compiling HTN plan verification problems into HTN planning problems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 145–150.

Lin, S.; Behnke, G.; Ondrčková, S.; Barták, R.; and Bercher, P. 2023. On total-order HTN plan verification with method preconditions–an extension of the CYK parsing algorithm. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 12041–12048.

Pantůčková, K.; and Barták, R. 2023. Using Earley Parser for Recognizing Totally Ordered Hierarchical Plans. In *Proceedings of 26th European Conference on Artificial Intelligence (ECAI) 2023*, 1819–1826. IOS Press.

Pantůčková, K.; Ondrčková, S.; and Barták, R. 2024. Using Earley Parser for Verification of Totally Ordered Hierarchical Plans. *The International FLAIRS Conference Proceedings*, 37(1).

Zaidins, P.; Roberts, M.; and Nau, D. 2023. Implicit Dependency Detection for HTN Plan Repair. *Proceedings of the 6th ICAPS Workshop on Hierarchical Planning (HPlan 2023)*, 10–18.