Modeling Assistance for Hierarchical Planning:  
An Approach for Correcting Hierarchical Domains with Missing Actions

Songtuan Lin¹, Daniel Höller², Pascal Bercher¹

¹School of Computing, The Australian National University  
²Computer Science Department, Saarland University  
{singtuan.lin, pascal.bercher}@anu.edu.au, hoeller@cs.uni-saarland.de

Abstract

The complexity of modeling planning domains is a major obstacle for making automated planning techniques more accessible, raising the demand of tools for providing modeling assistance. In particular, tools that can automatically correct errors in a planning domain are of great importance. Previous works have devoted efforts to developing such approaches for correcting classical (non-hierarchical) domains. However, no approaches exist for hierarchical planning, which is what we offer here. More specifically, our approach takes as input a flawed hierarchical domain together with a plan known to be a solution but actually contradicting the domain (due to errors in the domain) and outputs corrections to the domain that add missing actions to the domain which turn the plan into a solution. The approach achieves this by compiling the problem of finding corrections to another hierarchical planning problem.

Introduction

In the last few decades, a significant development of Automated Planning has been witnessed. Many techniques are developed for both hierarchical (Bercher, Alford, and Höller 2019) and non-hierarchical (Ghallab, Nau, and Traverso 2004) planning. In particular, tools that can automatically correct errors in a planning domain are of great importance. Previous works have devoted efforts to developing such approaches for correcting classical (non-hierarchical) domains. However, no approaches exist for hierarchical planning, which is what we offer here. More specifically, our approach takes as input a flawed hierarchical domain together with a plan known to be a solution but actually contradicting the domain (due to errors in the domain) and outputs corrections to the domain that add missing actions to the domain which turn the plan into a solution. The approach achieves this by compiling the problem of finding corrections to another hierarchical planning problem.
do not reduce the complexity of the problem because NP-completeness holds in TOHTN planning with only adding actions being allowed (Lin and Bercher 2021).

Our approach solves the problem of finding a minimal set of corrections to a domain by transforming it into another (total order) HTN planning problem such that a cost optimal solution to the transformed problem indicates the corrections. We will present how the encoding is done in the following sections. Along the way, we will also introduce some encoding techniques which can be used in broader scenarios, e.g., how to encode a counter in HTN planning.

Related Works

Significant effort has been devoted to developing tools for modeling support for non-hierarchical (classical) planning. Muise (2016) developed the successful online planning education platform, Planning.Domains, which supports a wide range of editing features such as syntax highlighting, auto-completion, and tuning an online planner. It provides interfaces for developing customized plugins for advanced features. Strobel and Kirsch (2020) developed a similar tool for education platforms, Planning.Domains, which supports a wide range of editing features such as stronger capability of syntax highlighting and automatic indentation. ItSIMPLE by Vaquero et al. (2013) is another widely used tool which eases modeling domain files such as stronger capability of syntax highlighting and automatic indentation. ItSIMPLE by Vaquero et al. (2013) is another widely used tool which eases modeling domain files such as stronger capability of syntax highlighting and automatic indentation. ItSIMPLE by Vaquero et al. (2013) is another widely used tool which eases modeling domain files such as stronger capability of syntax highlighting and automatic indentation. ItSIMPLE by Vaquero et al. (2013) is another widely used tool which eases modeling domain files such as stronger capability of syntax highlighting and automatic indentation.

Apart from those tools providing assistance restricted to supporting creating and editing domain files (i.e., on the syntax level), techniques have also been developed which provide advanced support (i.e., on the semantic level). Sreedharan et al. (2020) adopted the approach for explainable AI planning to correct a domain. This approach however works specifically for dialogue domains. Gragera et al. (2023) proposed an approach which fixes a domain by finding missing positive effects in actions. More concretely, they consider errors in a domain which cause a solvable problem becoming unsolvable. Hence, their approach takes as input an unsolvable hierarchical planning problem and outputs corrections to the domain by turning all those plans into solutions. We will present how the encoding is done in the following sections. Along the way, we will also introduce some encoding techniques which can be used in broader scenarios, e.g., how to encode a counter in HTN planning.

Significant effort has been devoted to developing tools for modeling support for non-hierarchical (classical) planning. Muise (2016) developed the successful online planning education platform, Planning.Domains, which supports a wide range of editing features such as syntax highlighting, auto-completion, and tuning an online planner. It provides interfaces for developing customized plugins for advanced features. Strobel and Kirsch (2020) developed a similar tool for education platforms, Planning.Domains, which supports a wide range of editing features such as stronger capability of syntax highlighting and automatic indentation. ItSIMPLE by Vaquero et al. (2013) is another widely used tool which eases modeling domain files such as stronger capability of syntax highlighting and automatic indentation. ItSIMPLE by Vaquero et al. (2013) is another widely used tool which eases modeling domain files such as stronger capability of syntax highlighting and automatic indentation. ItSIMPLE by Vaquero et al. (2013) is another widely used tool which eases modeling domain files such as stronger capability of syntax highlighting and automatic indentation. ItSIMPLE by Vaquero et al. (2013) is another widely used tool which eases modeling domain files such as stronger capability of syntax highlighting and automatic indentation.
A method $m = (c, tn)$ decomposes a compound task $c \in C$ into a task network $tn$ which is a sequence of primitive and compound tasks (i.e., $tn \in (A \cup C)^*$ where * is the Kleene star), written as $c \rightarrow_m tn$. The notion of decomposition can be extended from a single compound task to a task network. Concretely, let $tn = (t_1 \cdots t_n)$ be a task network in which $n \in \mathbb{N}$ and $t_k \in A \cup C$ for each $1 \leq k \leq n$ and $m = (t_i, tn^*)$ a method with $tn^* = (t_{i_1}^* \cdots t_{i_p}^*)$ that decomposes a task $t_i$ ($1 \leq i \leq n$) in $tn$ into the task network $tn^*$, we say that $tn$ is decomposed into another task network $tn'$ by $m$ iff

$$tn' = (t_1 \cdots t_{i-1} t_{i_j}^* \cdots t_{i_p}^* t_{i+1} \cdots t_n)$$

That is, $t_i$ in $tn$ is replaced by $tn^*$. For any compound task $c$ or task network $tn$, we write $c \rightarrow_m^* tn'$ (resp. $tn \rightarrow_m^* tn'$) to denote that $c$ (resp. $tn$) is decomposed into the task network $tn'$ by a sequence of methods $m$.

A solution to an HTN planning problem is a primitive task network (i.e., an action sequence) $\pi = \langle a_1 \cdots a_n \rangle$ such that there is a method sequence $m = c \rightarrow_m^{\pi} \pi$, and $s_1 \rightarrow^* s$ for some $s$ with $q \subseteq s$ (i.e., the goal is satisfied in $s$). In many scenarios, an action $a$ in a planning problem has certain cost. The total cost of an action sequence $\pi$ is the summation of the cost of every action in it. A cost-optimal solution to a planning problem is a solution of the minimal cost, i.e., there exist no other solutions of cost smaller than it.

Fig. 1 shows an example about an HTN planning problem. Each white box is a compound task while each blue box represents a primitive one. The problem has four propositions, \{p, q, f, r\} and four actions, \{a_1, \ldots, a_4\}. The preconditions and effects of each action are depicted in the figure. The initial task $c_1$ can be decomposed by solely one method $m_1$ into a sequence of two compound tasks, $\langle c_1 \ c_2 \rangle$. $c_1$ can be decomposed into an action sequence $\langle a_1 \ a_2 \rangle$ by the method $m_1$ while $c_2$ can be decomposed into either $\langle a_3 \rangle$ or $\langle a_4 \rangle$ by the method $m_2$ or $m_3$, respectively. The initial state $s_1 = \{p\}$. The goal $g$ is $\{r\}$. The action sequence $\langle a_1 \ a_2 \ a_3 \rangle$ is a solution as it can be obtained by decomposing $c_1$ using the method sequence $\langle m_1 \ m_1 \ m_3 \rangle$, and the action sequence is executable and achieves the goal. Note that $\langle a_1 \ a_2 \ a_4 \rangle$ is not a solution despite that it can also be obtained by decomposing $c_1$. It is not executable because the precondition of $a_4$ is not satisfied ($p$ is removed by $a_1$).

One remark regarding the example is that the method sequence $\langle m_1 \ m_2 \ m_1 \rangle$ can also decompose the initial task $c_1$ into $\langle a_1 \ a_2 \ a_3 \rangle$. In fact, both two method sequences which results in the solution are captured by the same decomposition hierarchy shown in Fig. 1. Such a hierarchy is called a decomposition tree (Geier and Bercher 2011), which shows how a solution is obtained by decomposing the initial task. One may further notice that for TOHTN planning, a decomposition tree shares a lot of similarities with a parsing tree in the context of context-free grammars (CFGs). Höller et al. (2014) have shown that a TOHTN planning problem in fact is identical to a CFG in the sense that we could view a primitive task as a terminal symbol, a compound task as a non-terminal symbol, and a method as a production rule.

Having presented the planning formalism, we now formulate the problem of correcting an HTN planning domain. For this purpose, we first define atomic corrections with respect to a domain. Since we make the assumption that flaws in a domain stem from missing actions in methods, we thus only consider corrections that add actions to methods. Formally, let $D$ be a domain and $m \in M$ an arbitrary method with $m = (c, tn)$ and $tn = (t_1 \cdots t_n)$ for some $n \in \mathbb{N}$, we define $I[a, m, i] \leftrightarrow a \in A$ and $0 \leq i \leq n$ as a correction which inserts the action $a$ into the position between $t_i$ and $t_{i+1}$ in $m$. As two special cases, when $i = 0$ or $n$, the action $a$ will simply be inserted into the position before $a_1$ or after $a_n$. As an example, applying the correction $I[a, m, i]$ will modify the method $m$ into a new one which decomposes the compound task $c$ into the task network $\langle t_1 \cdots t_i \ a \ t_{i+1} \cdots t_n \rangle$. Let $o$ be an atomic correction with respect to a domain $D$, we use the notation $D \Rightarrow_o D'$ to indicate that $D'$ is a new domain obtained by applying $o$ to $D$. The problem of correcting an HTN domain is then defined as follows.

**Definition 1.** Let $\Pi = ( D, s_I, c_I, g )$ be an HTN planning problem and $\pi = \langle a_1 \cdots a_n \rangle$ an action sequence, the domain correction problem is a tuple $(\Pi, \pi)$ which is to find an optimal sequence of corrections $\langle o_1 \cdots o_j \rangle$ such that $D \Rightarrow_o D_1 \Rightarrow_o \cdots \Rightarrow_o \cdots \Rightarrow_o D_{j-1} \Rightarrow_o D_j$ and $\pi$ is a solution to the planning problem $\Pi'$ with $\Pi' = ( D_j, s_I, c_I, g )$. In particular, by a correction sequence being optimal, we mean that there exists no other correction sequences of smaller length which can turn the action sequence into a solution.

Note that in this paper, we do not consider corrections to actions’ preconditions and effects (flaws in actions’ preconditions and effects might cause an action sequence not being executable and hence not being a solution). This is however equivalent to correcting a non-hierarchical domain (one could identify that correcting actions’ preconditions and effects is orthogonal to correcting methods) and has been addressed properly (Lin, Grastien, and Bercher 2023).

**Encoding**

Now we shift our attention to solving the problem of correcting an HTN domain. We do so by transforming this problem into an HTN planning problem. Intuitively speaking, the transformed problem completes two tasks: 1) It decides what corrections should be made to the domain, and 2) it verifies
whether the given action sequence is a solution to the planning problem with the new domain. Note that the latter one is the plan verification problem for HTN planning from which a reduction to an HTN planning problem (Höller et al. 2022) exists. Hence, our encoding is built on top of theirs while incorporating corrections to the domain. For clarity, we start by reproducing the transformation from the plan verification problem to an HTN planning problem, and then we introduce how to incorporate corrections into it.

Encoding the Plan Verification Problem

Consider an HTN planning problem $\Pi = (D, s_I, c_I, g)$ and an action sequence $\pi = (a_1 \ldots a_n)$. Note that $\pi$ could have duplicate actions, i.e., there exist some $1 \leq i, j \leq n$ with $i \neq j$ and $a_i = a_j$. For simplicity, since we do not consider correcting actions’ preconditions and effects, we assume that for any action $a \in A$, $\text{proc}(a) = \text{eff}^+(a) = \text{eff}^-(a) = \emptyset$, and $g = s_I = P = \emptyset$. The goal of the encoding is to construct a new HTN planning problem $\Pi'$ such that $\Pi'$ has a solution if and only if $\pi$ is a solution to $\Pi$.

By consulting the solution criteria for HTN planning, one can observe that the core of deciding whether $\pi$ is a solution to $\Pi$ is searching for a decomposition hierarchy (a method sequence) decomposing $c_I$ to $\pi$. To simulate this search procedure, the constructed problem $\Pi'$ should preserve the following two properties: 1) $\Pi'$ has only one solution $\pi'$, encoding $\pi$, and 2) the decomposition hierarchy decomposing $c_I$ into $\pi'$ simulates the one that decomposes $c_I$ into $\pi$.

To ensure the former, consider an action $a$ in $\Pi$ with $a = a_i$ for some $a_i$ in $\pi$. A primitive task $a_i'$ is constructed for $\Pi'$, encoding that $a$ appears at the $i$th position of $\pi$. For convenience, we use $A[a, i]$ to denote $a_i'$. One could think of $A[a, i]$ as a function which takes as input a primitive action $a$ in $\Pi$ and a position $i$ in $\pi$ and outputs the respective action constructed for $\Pi'$. For convenience, throughout the paper, we will use $A[a, i]$, $C[a]$, $M[a]$, and $P[a]$ to represent the action, the compound task, the method, and the proposition constructed according to certain parameters, respectively. In other words, for each action $a$ in $\Pi$, a set $A[a]$ of primitive tasks is constructed for $\Pi'$ such that

$$A[a] = \{ A[a, i] \mid a = a_i \text{ for some } a_i \text{ in } \pi \}.$$

Here we abuse the notation to let $A[a]$ denote the action set. Furthermore, in order to make the semantics of $A[a, i]$ hold, additional propositions shall be constructed as $A[a, i]$’s precondition and effects. More specifically, the action $A[a, i]$ has solely one precondition $P[a, i] = \{ i - 1 \}$, one positive effect $P[a, i]$, and one negative effect $P[a, i] = \{ i - 1 \}$ where the parameters refer to the respective positions in $\pi$. The positive effect $P[a, i]$ asserts that $A[a, i]$ occupies the $i$th position of $\pi$ while the precondition $P[a, i] = \{ i - 1 \}$ ensures that the $(i - 1)$th position of $\pi$ must already be occupied. The negative effect $P[a, i] = \{ i - 1 \}$ ensures that the $(i + 1)$th position must be the next one to be occupied.

By letting $s_I' = \langle P[0], \ldots, P[n] \rangle$ and $g' = \{ P[a] \}$, we ensure that the sole solution $\pi'$ to $\Pi'$ is that $\pi' = \langle A[a_1, 1] \ldots A[a_n, n] \rangle$.

To ensure the second property, $\Pi'$ preserves all compound tasks and methods in $\Pi$ except that for each existing method $m$, every action $a$ in $m$ is replaced with a newly constructed compound task. That is, for each $a \in A$ in $\Pi$, a compound task denoted as $C[a]$ is constructed for $\Pi'$. One could again view $C[a]$ as a function which maps the action $a$ in $\Pi$ to the respective compound task in $\Pi'$. For convenience, we also define $C[c] = c$ for each $c \in C$ in $\Pi$, meaning that $c$ is preserved in $\Pi'$. Furthermore, each $C[a]$ can be decomposed by $A[a]$ methods each of which decomposes $C[a]$ into an action $A[a, i]$, $i \in A[a]$ in $\Pi$. Similarly, we use $M[m]$ to denote the method in $\Pi'$ which adheres to the method $m$ in $\Pi$ and $M[a, i]$ to denote the one decomposing $C[a]$ into $A[a, i]$.

By construction, if a sequence of methods $(m_1 \ldots m_j)$ for $\Pi$ decomposes $c_I$ into $\pi$, the sequence $(\langle M[m_1], \ldots, M[m_j] \rangle)$ then decomposes $C[c_I]$ into $(C[a_1] \ldots C[a_n])$, which can further be decomposed into $(A[a_1, 1] \ldots A[a_n, n])$, meaning that $\Pi'$ has a solution if and only if $\pi$ is a solution to $\Pi$. For the detailed proof for this fact, see the work by Höller et al. (2022).

In summary, the problem $\Pi' = (D', s_I', c_I', g')$ with $D' = \langle P', A', C', M', \alpha' \rangle$ is constructed as follows:

- $P' = \{ P[0], \ldots, P[n] \}$, and $A' = \bigcup_{a \in A} A[a]$.
- $C' = \{ C[t] \mid t \in A \cup C \}$.
- $M' = \{ M[m] \mid m \in M \} \cup M^+$ with $M^+$ being the set $\{ M[a, i] \mid a = a_i \text{ for some } a_i \text{ in } \pi \}$.
- $s_I' = \{ P[0], \ldots, P[n] \}$, and $g' = \{ P[a] \}$, and $c_I' = C[c_I]$.

Fig. 2 shows an example about how to transform a plan verification problem to an HTN planning problem.

Incorporating Corrections

Having presented how to encode the plan verification as an HTN planning problem, we now move on to incorporate domain corrections into the encoding, which is our main contribution.
Let $\Pi' = (D', s'_t, c'_t, g'_t)$ with $D' = (P', A', C', M', \alpha')$ be the HTN problem which encodes the problem of deciding whether a plan $\pi = (a_1 \cdots a_n)$ is a solution to an HTN problem $\Pi = (D, s_t, c_t, g)$ with $D = (P, A, C, M, \alpha)$. Our goal is to make some further constructions to $\Pi'$ to simulate making corrections to $\Pi$. Since we only consider adding missing actions, our new constructions are thus restricted to methods $M[m]$ in $M'$ with $m \in M$. This is because $M[m]$ is basically a copy of $m$, and hence, adding actions to $M[m]$ already simulates adding actions to $m$.

Consider a method $M[m]$ decomposing a compound task into a task network $(t_1 \cdots t_k)$. We first observe that there are $k + 1$ blocks to which we can add actions, namely, any block between the tasks $t_{i-1}$ and $t_i$ for some $1 < i \leq k$ together with the two that are before $t_1$ and after $t_k$. The fundamental idea of simulating action insertions is constructing one compound task for each such blocks, controlling which actions should be added to the respective block. We use $b(m', i)$ with $m' = M[m]$ to denote a block (i.e., the $i$th block in the method $m'$) and $C[m', i]$ to denote the compound task constructed for the respective block.

Concretely, each $C[m', i]$ is placed at the respective block in the method $m'$. The control over action insertions is done by decomposing $C[m', i]$ into a task network that consists of $n$ compound tasks, $(C[m', i, 1] \cdots C[m', i, n])$, where each $C[m', i, j]$ $(1 \leq j \leq n)$ is to decide whether the action $a_j$ in $\Pi$ is inserted into the block $b(m', i)$. We use $M[m', i]$ to denote the method that decomposes $C[m', i]$. Fig. 3 depicts an example about how to revive a method $m$ with $m' = M[m]$ with $m \in M$ for some $m$, $m'$ initially contains two tasks (represented as solid boxes), meaning that there are three blocks into which actions can be inserted. We thus construct three compound tasks $C[m', i]$ with $0 \leq i \leq 2$, and put them into the respective places.

In order to encode the situation that $a_j$ is added to $b(m', i)$, we construct a method $M[m', i, j, +]$ which decomposes $C[m', i, j]$ into the compound task $C[a_j]$. Intuitively speaking, the reason for using $C[a_j]$ here is that the action $a_j$ in $\Pi$ is represented as $C[a_j]$ in $\Pi'$. We will explain this in more detail later on. Similarly, in order to encode that $a_j$ is not inserted, we construct another method $M[m', i, j, -]$ which decomposes $C[m', i, j]$ into an empty task network.

Methods $M[m', i, j, \{+,-\}]$ with $m' = M[m]$ for some method $m$ in $\Pi$ are however not constructed adequately. The reason is that in the scenario of correcting the domain of $\Pi$, if an action is inserted to the method $m$ which occurs more than once in decomposition, the insertion must take effects for all occurrences of $m$. The analogy of this constraint for $\Pi'$ is that the consequence of adding (or not adding) an action to $m'$ must be consistent among all occurrences of $m'$.

To achieve such consistency, we construct one more primitive task $A[m', i, j, +]$ for $M[m', i, j, +]$ and put it in front of the task $C[a_j]$. Similarly, we construct another primitive task $A[m', i, j, -]$ and place it into $M[m', i, j, -]$. The action $A[m', i, j, +]$ has one precondition, $P[m', i, j, +]$, one negative effect, $P[m', i, j, -]$, and no positive effects. In contrast, $A[m', i, j, -]$ has $P[m', i, j, -]$ as its precondition while it deletes $P[m', i, j, +]$. We put these two propositions into the initial state. By construction, if for the first occurrence of $m'$ in some method sequence, $M[m', i, j, +]$ is applied to decompose $C[m', i, j]$ (which encodes adding $a_j$ to the block $b(m', i)$), then for other occurrences of $m'$, $M[m', i, j, -]$ is not available because the precondition of $A[m', i, j, -]$ has been removed, ensuring the consistency.

The construction of these actions as well as the structure of the methods $M[m', i, j, \{+,-\}]$ is shown in Fig. 4. At this moment, one could ignore the proposition $P'$, the action $A'$, and the method $M'$ in Fig. 4. We will describe the construction and the purpose of those components shortly. The compound task $C[\mathcal{U}]$ in the method $M[m', i, j, +]$ represents a counter which can count up to a certain bound $\mathcal{U}$. The purpose of the counter is to make our constructed HTN problem more easy to be solved. The detailed implementation of the counter will be introduced in the next section.

The fact that the method $m'$ can be used more times than once in decomposition is another reason for why using $C[a_j]$ in $M[m', i, j, +]$. The action $a_j$ inserted might also be used to match another action $a_k$ in $\Pi$ with $a_k = a_j$ and $k > j$. For such a case, $C[a_j]$ can be decomposed into $A[a_j, a_j]$ for the first occurrence of $m'$ and $A[a_k, k]$ for the other.

Taken together, the HTN problem $\Pi' = (D', s'_t, c'_t, g'_t)$ with $D' = (P', A', C', M', \alpha')$ which encodes the problem of correcting an HTN domain is as follows:

- $P' = P' \cup P^1$ where $P^1$ is the set consisting of propositions $P[m', i, j, o]$ with $m' = M[m]$ for some $m \in M$, $1 \leq j \leq n$, $o \in \{+,-\}$, and $0 \leq i \leq |m'| + 1$. Here, $|m'|$ is the length of the task sequence resulting from $m'$.
- $A'$ is the union of $A'$ and $A'$ where $A'$ is the set of actions $A[a'_m, i, j, o]$ whose parameters are under the same constraints as those of $P[m', i, j, o]$ defined above.
- $C' = C' \cup B \cup S$ where $B$ is the set of compound tasks $C[m', i]$ and $S$ consisting of tasks $C[m', i, j]$, $m', i$, and $j$ all follow the same constraints as above.
- $M' = M' \cup M'^1 \cup M'$ where $M'^1$ is the set of methods each of which decomposes a compound task $C[m', i, j]$ and $M'$ contains the methods $M[m', i, j, o]$ with the parameters following the same constraints as above.
- $s'_t = s'_t \cap I$ with $I = P', c'_t = c'_t$, and $g'_t = g'_t$.

The construction ensures that the plan $\pi$ can be turned into a solution to $\Pi'$ by correcting the domain of $\Pi'$ that decomposes $\pi'$ into a solution. Methods $M[m', i, j, +]$ and $M[m', i, j, -]$ for any
The number of actions inserted to $\Pi^*$ has been used previously. Since $\Pi^*$ indicates the number of action insertions allowed. If we could incorporate this bound into our construction, an HTN problem solver (which is used to solve the constructed problem) can exploit this information to significantly reduce the search space by not adding any further action when the number of insertions has reached the bound. One could immediately observe that given a domain correction problem $(\Pi, \pi)$ with $\Pi = (D, s_I, c_I, g)$ and $\pi = \langle a_1 \cdots a_n \rangle$, the maximal number $U$ of action insertions to $D$ is $n - \gamma(c_I)$ where $\gamma(c_I)$ represents the minimal number of actions that can be obtained from $c_I$. For if we add more actions than that bound, then any primitive task network obtained from the initial task has actions which outnumber the given plan $\pi$, meaning that $\pi$ can never be obtained.

**Theorem 1.** $\Pi^*$ is solvable iff $\pi$ can be turned into a solution to $\Pi$ by adding actions to methods in $\Pi$.

However, we are looking for an optimal sequence of corrections. To this end, we want to assign cost to the actions in $\Pi^*$ such that the cost optimal solution to $\Pi^*$ indicates the optimal corrections. One naive attempt is letting each $A \langle m', i, j, + \rangle$ have cost one, representing the cost of adding an action. The remaining actions all have zero cost. This is however not enough. If an action is added to a method $m'$ while $m'$ is applied multiple times, $A \langle m', i, j, + \rangle$ associated with this insertion will occur multiple times, meaning that the cost of this insertion is also counted more than once.

To address this problem, for each $M \langle m', i, j, + \rangle$, we construct a new method $M' \langle m', i, j, + \rangle$ which is identical to the original one except that the action $A \langle m', i, j, + \rangle$ is replaced by a new one $A' \langle m', i, j, + \rangle$. This action has zero cost. Both positive and negative effects of it is empty. Its sole precondition is a new proposition $P' \langle m', i, j, + \rangle$, which is also added to the positive effects of $A \langle m', i, j, + \rangle$. As a result, the method $M' \langle m', i, j, + \rangle$ can be applied iff $M \langle m', i, j, + \rangle$ has been used previously. Since $A' \langle m', i, j, + \rangle$ has zero cost, this construction ensures that the cost of every action insertion is counted only once. Consequently, the cost optimal solution to the problem $\Pi^*$ with this new construction reveals the optimal correction to the domain of $\Pi$.

**Counter**

The construction presented previously already encodes the domain correction problem. We could further improve it by providing the maximal number of action insertions allowed. To address this problem, for each $M \langle m', i, j, + \rangle$, we construct a new method $M' \langle m', i, j, + \rangle$ which is identical to the original one except that the action $A \langle m', i, j, + \rangle$ is replaced by a new one $A' \langle m', i, j, + \rangle$. This action has zero cost. Both positive and negative effects of it is empty. Its sole precondition is a new proposition $P' \langle m', i, j, + \rangle$, which is also added to the positive effects of $A \langle m', i, j, + \rangle$. As a result, the method $M' \langle m', i, j, + \rangle$ can be applied iff $M \langle m', i, j, + \rangle$ has been used previously. Since $A' \langle m', i, j, + \rangle$ has zero cost, this construction ensures that the cost of every action insertion is counted only once. Consequently, the cost optimal solution to the problem $\Pi^*$ with this new construction reveals the optimal correction to the domain of $\Pi$.

**Theorem 1.** $\Pi^*$ is solvable iff $\pi$ can be turned into a solution to $\Pi$ by adding actions to methods in $\Pi$.

However, we are looking for an optimal sequence of corrections. To this end, we want to assign cost to the actions in $\Pi^*$ such that the cost optimal solution to $\Pi^*$ indicates the optimal corrections. One naive attempt is letting each $A \langle m', i, j, + \rangle$ have cost one, representing the cost of adding an action. The remaining actions all have zero cost. This is however not enough. If an action is added to a method $m'$ while $m'$ is applied multiple times, $A \langle m', i, j, + \rangle$ associated with this insertion will occur multiple times, meaning that the cost of this insertion is also counted more than once.

To address this problem, for each $M \langle m', i, j, + \rangle$, we construct a new method $M' \langle m', i, j, + \rangle$ which is identical to the original one except that the action $A \langle m', i, j, + \rangle$ is replaced by a new one $A' \langle m', i, j, + \rangle$. This action has zero cost. Both positive and negative effects of it is empty. Its sole precondition is a new proposition $P' \langle m', i, j, + \rangle$, which is also added to the positive effects of $A \langle m', i, j, + \rangle$. As a result, the method $M' \langle m', i, j, + \rangle$ can be applied iff $M \langle m', i, j, + \rangle$ has been used previously. Since $A' \langle m', i, j, + \rangle$ has zero cost, this construction ensures that the cost of every action insertion is counted only once. Consequently, the cost optimal solution to the problem $\Pi^*$ with this new construction reveals the optimal correction to the domain of $\Pi$.

**Counter**

The construction presented previously already encodes the domain correction problem. We could further improve it by providing the maximal number of action insertions allowed. If we could incorporate this bound into our construction, an HTN problem solver (which is used to solve the constructed problem) can exploit this information to significantly reduce the search space by not adding any further action when the number of insertions has reached the bound. One could immediately observe that given a domain correction problem $(\Pi, \pi)$ with $\Pi = (D, s_I, c_I, g)$ and $\pi = \langle a_1 \cdots a_n \rangle$, the maximal number $U$ of action insertions to $D$ is $n - \gamma(c_I)$ where $\gamma(c_I)$ represents the minimal number of actions that can be obtained from $c_I$. For if we add more actions than that bound, then any primitive task network obtained from the initial task has actions which outnumber the given plan $\pi$, meaning that $\pi$ can never be obtained.

**Proposition 1.** Let $(\Pi, \pi)$ be a domain correction problem. The number of actions inserted to $\Pi$ turning $\pi$ into a solution cannot exceed $|\pi| - \gamma(c_I)$ where $|\pi|$ is the length of $\pi$, $c_I$ is the initial task of $\Pi$, and $\gamma(c_I)$ is the minimal number of actions that can be obtained from $c_I$.
has one single positive effect \( \mathbb{P}[i, \mathcal{U}] \), indicating that the step \( i \) has been counted. \( \mathcal{A}[i, \mathcal{U}] \) also removes \( \mathbb{P}[i - 1, \mathcal{U}] \), which ensures that the counter can only count incrementally. Fig. 5 illustrates the implementation of a counter.

To incorporate the bound \( \mathcal{U} \) into the construction, we simply place the counter \( \mathcal{C}[\mathcal{U}] \) at the beginning of every method \( \mathcal{M}[m', i, j, +] \). This thus ensures that when an action is inserted, the counter will increase by 1, and when the counter reaches \( \mathcal{U} \), no further methods \( \mathcal{M}[m', i, j, +] \) can be applied, i.e., no further actions can be inserted. Note that the counter \( \mathcal{C}[\mathcal{U}] \) is not added to methods \( \mathcal{M}[m', i, j, +] \), ensuring that one action insertion will not be counted multiple times.

**Empirical Evaluation**

In this section, we present the experimental results with respect to our approach. The two metrics we used to evaluate the performance of the approach are the runtime for correcting a domain and the coverage on the benchmark set (i.e., the percentage of the instances that can be solved in the benchmark set). The reason for using these two is that in the real scenario of modeling support, the time required for finding corrections is the most critical factor because when deploying our approach into practice, it can be called iteratively and interactively. In every iteration, the user can decide whether the found corrections address the errors successfully. If that is not the case, the user can rule out those unsatisfactory corrections and instruct our approach to block them in the next iteration. This continues until all errors are addressed. Note that we did not compare our approach with the one by Xiao et al. (2020) because, as mentioned in the related work section, the inputs to these two approaches are different.

**Configuration**

The experiments ran on an Intel Cascade Lake CPU, with 20-minutes timeout. However, there existed no benchmark sets of flawed hierarchical domains at the time when we conducted the empirical evaluation, and hence, we had to construct a novel benchmark set. The procedure for constructing the benchmark set is as follows. We drew 200 TOHTN planning problem instances together with the respective solutions from 11 domains from the IPC 2020 on HTN Planning benchmark set. We emphasize that the meaning of the term “domain” here is different from a domain \( \mathcal{D} \) defined in the TOHTN planning formalism. The term “domain” here refers to the name of a class of HTN planning problems that model problems in the same scenario. For instance, the domain *Rover* refers to the class of planning problems in the scenario of Mars exploration.

Next, for each drawn problem instance \( \Pi = (\mathcal{D}, s_1, c_1, g) \) with \( \mathcal{D} = (\mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{M}, \alpha) \), we let every action in every method \( m \in \mathcal{M} \) have a 30% chance of being removed. This thus randomly introduced errors to the domain \( \mathcal{D} \). Lastly, we discarded those instances to which no errors were introduced. This left 167 instances with flawed domains in total.

**Experimental Results**

The experimental results (Lin, Höller, and Bercher 2024) are summarized in Tab. 1. The left-most column lists the domain names. The columns labeled with “total” and “solved” show the total number of instances and of solved instances. The two columns under the name “plan length” report the minimum and the maximal length of provided plans. As can be seen from the table, our approach solved around 37.12% instances, i.e., it can provide optimal corrections to 37.12% flawed domains. Our approach performed badly on the domains Hiking and Transport. For the former, the reason is that the domain already contains numerous methods. Our approach thus creates an HTN problem of large size which is hard to be solved by an HTN planner. For the Transport domain, it is not clear what causes the bad performance. We hypothesize that it is due to some specific structure, e.g., cyclic structure, within the domain.

Fig. 6 shows the runtime for solving each instance against the plan length, including those unsolved instances whose runtime exceeded the timeout, i.e., each point in the plot represents a domain correction problem instance.

**Discussions and Future Work**

In this paper we assume that errors in a domain only attribute to missing actions, causing the limitation of our approach that corrections are restricted to inserting actions. In practice, there are more types of corrections that could be considered, e.g., adding methods/compound tasks and deleting actions/methods/compound tasks. One remark is that inserting
actions can serve as a basis for many of those corrections. E.g., the treatment for inserting compound tasks is similar to inserting actions while adding methods can be viewed as a two-step process: We first create an empty method and then insert actions and compound tasks into it. Consequently, we plan in our future work to implement all those corrections so that our approach can be deployed more broadly.

Additionally, our approach only works for grounded HTN planning. The lifted formalism is however more widely used in the context of domain modeling. The grounded formalism is defined in terms of propositional logic whereas the lifted one is defined in first-order logic. Hence, the lifted formalism is a generalization of the grounded one, meaning that all corrections which are meaningful in the grounded setting are also relevant to the lifted one whereas the lifted formalism can have additional corrections, e.g., correcting the arguments of a primitive task, a compound task, or a method. An important future work is thus to generalize our approach so that it can work for the lifted setting.

We have mentioned before that correcting primitive tasks’ preconditions and effects is also an important branch of correcting a planning domain. This task has been addressed by Lin, Grastien, and Bercher (2023). However, there exist hierarchical planning formalisms which support methods/compound tasks’ preconditions and effects. Fixing these components is also an important aspect of fixing domains.

Another limitation is that some corrections returned might not be desired by the modeler. To address this problem, we want to employ an iterative process involving human-in-the-loop. Concretely, our approach is invoked iteratively. On each iteration, the user could decide which corrections returned by our approach are desired and which are not. The next iteration will then start by adapting those desired corrections and forbidding those undesired ones. This process keeps running until the domain is completely fixed.

Our evaluation shows that our approach only solved less than 50% problems in the evaluation. To improve the performance, we consider transforming the problem to SAT or ILP because they have the same complexity as the domain correction problem, i.e., NP-complete (whereas TOHTN planning is EXPTIME-complete (Alford, Bercher, and Aha 2015)), which might have more efficient solving techniques given the specific problem.

Conclusion

We presented an approach for correcting a TOHTN domain with missing actions by compiling such a problem to a planning problem. We are the first to study such an approach for hierarchical planning, making it an important contribution toward modeling supports for hierarchical planning. Our approach can also be applied to a wide range of applications because of the connection between a TOHTN problem and a context-free grammar.

References


