HTN Plan Repair Algorithms Compared: Strengths and Weaknesses of Different Methods *

Paul Zaidins[1], Robert P. Goldman[2], Ugur Kuter[2], Dana Nau[1], Mark Roberts[3]

[1]Department of Computer Science and Institute for Systems Research, University of Maryland
[2]SIFT, LLC
[3]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
pzaidins@umd.edu, rpgoldman@sift.net, ukuter@sift.net, nau@umd.edu, mark.c.roberts20.civ@us.navy.mil

Abstract

This paper provides theoretical and empirical comparisons of three recent hierarchical plan repair algorithms: SHOPFIXER, IPYHOPPER, and REWRITE. Our theoretical results show that the three algorithms correspond to three different definitions of the plan repair problem, leading to differences in the algorithms' search spaces, the repair problems they can solve, and the kinds of repairs they can make. Understanding these distinctions is important when choosing a repair method for any given application.

Building on the theoretical results, we evaluate the algorithms empirically in a series of benchmark planning problems. Our empirical results provide more detailed insight into the runtime repair performance of these systems and the coverage of the repair problems solved, based on algorithmic properties such as replanning, chronological backtracking, and backjumping over plan trees.

1 Introduction

Dynamic control and management of plans during execution is challenging in many multi- and hierarchically-organized agent systems, e.g., military operations, warehouse automation, and other practical applications. During execution, agents' plans may fail due to precondition failures induced by exogenous events or by actions of other agents. Fox et al. (2006) showed that in the face of disruptions, plan repair could provide new plans faster and with fewer revisions than replanning from scratch. They used the term "stability" to refer to the new plan's similarity to the old one, by analogy to the term from control theory.

Agent systems using Hierarchical Task Network (HTN) planning generate by decomposing high-level (more abstract) tasks into lower-level (less abstract) subtasks before or during execution. Generalizing stable plan repair from classical planning to HTN planning requires localization of disruptions, errors and failures in the hierarchies, and it uses problem refinement methods that take advantage of such localizations to provide better stability (Goldman, Kuter, and Freedman 2020). Early work on hierarchical plan repair introduced validation graphs to hierarchical partial-order planning, using the validation graphs to identify disruptions and make patches in the partial-order plans (Kambhampati and Hendler 1992). Several recent algorithms, namely SHOPFIXER (Goldman, Kuter, and Freedman 2020), IPYHOPPER (Zaidins, Roberts, and Nau 2023), and an unnamed algorithm that we will call REWRITE (Höller et al. 2020b), have developed state-of-the art plan-repair strategies. They build on several previous methods (Ayan et al. 2007; Kuter 2012; Bansod et al. 2022; Bercher et al. 2014).

This paper analyzes the formal properties and performance of IPYHOPPER, SHOPFIXER, and REWRITE. We compare them formally, and empirically evaluate their performance in a series of benchmark planning problems. The following paragraphs summarize our contributions:

First, to enable comparison with REWRITE's definition of plan repair (Höller et al. 2020b), we formally define the notions of plan repair used by IPYHOPPER and SHOPFIXER. This enables us to prove the following results:

1. The definitions correctly characterize the search spaces of IPYHOPPER and SHOPFIXER.
2. The sets of solutions for the three algorithms are distinct but not disjoint.
3. We prove other relationships among the sets of solutions.

Second, we empirically evaluate algorithm performance on three domains drawn from the International Planning Competition (IPC): Openstacks, Rovers, and Satellites. Here are our primary empirical findings:

4. REWRITE is often the slowest algorithm, because it extensively rederives plans.
5. The causal links and backjumping of SHOPFIXER are useful for larger problems in improving performance by narrowing the size of the search space.
6. The chronological backtracking and simple forward simulation of IPYHOPPER can outperform more sophesticated methods for smaller problems due to its minimal overhead.

2 Preliminaries

We begin with the usual classical definitions (Ghallab, Nau, and Traverso 2004), e.g., a state s is a set of ground atoms, an action a's preconditions are $\mathrm{pre}(a)$ (in the general case,

*Figure titles of the form of "Figure S#" refer to figures in the supplemental material at http://arxiv.org/abs/2504.16209.

a First Order Logic formula), and if s satisfies $\mathrm{pre}(a)$ then a is applicable in s and $\gamma(s, a)$ is the resulting state. A *plan* is a possibly-empty action sequence $\pi = \langle a_1, \ldots, a_n \rangle$. Applicability of π in a state s_0, and the state $\gamma(s_0, \pi)$ produced by applying it, are defined in the usual way.

To denote the parts of π before and after a_i, we will write $\pi[\prec a_i] = \langle a_1, \ldots, a_{i-1} \rangle$ and $\pi[\succ a_i] = \langle a_{i+1}, \ldots, a_n \rangle$.

An *HTN planning domain* is a pair $\Sigma = (\Sigma_c, \mathcal{M})$, where Σ_c is a classical planning domain and $\mathcal{M}$ is a set of methods (see (Ghallab, Nau, and Traverso 2004) for details). We will only consider *total-order* HTN planning, in which every method's subtasks are totally ordered. We let M be the set of all ground instances of the methods in $\mathcal{M}$.

A *decomposition tree* (d-tree), T, represents a recursive decomposition of a task or a task sequence. For a single task t, T has a root node labeled with t. For a task sequence $\tau = \langle t_1, \ldots, t_n \rangle$, T has an empty root node whose children are d-trees for the tasks $t_1, \ldots, t_n$. The primitive tasks (i.e., actions) in T have no children. Each nonprimitive task t' in T has a child node labeled with a ground method that is relevant for t'. Each ground method m in T has a sequence of child nodes labeled with m's subtasks, or if m has no subtasks then it has one child node labeled by a dummy action with no preconditions and no effects (this simplifies some of the recursive definitions in the next section).

We represent T's subtrees with the following notation. $T[t]$ is the subtree having t as its root; $T[\prec t]$ is the subtree containing t's postorder predecessors (i.e., nodes before t in a postorder traversal); $T[\preceq t]$ is the subtree containing both $T[\prec t]$ and t; and $T[\succ t]$ and $T[\succeq t]$ work similarly.

An *HTN planning problem* is a triple $P = (\Sigma, s_0, \tau_0)$, where Σ is the HTN planning domain, s_0 is the initial state, and $\tau_0 = \langle t_1, \ldots, t_n \rangle$ is a (possibly empty) sequence of tasks to accomplish. A *solution tree* for P is a decomposition tree for τ_0 that is applicable in s_0.

Suppose we have executed part of a solution tree T for P. Let $\pi = \mathrm{plan}(T)$ be the sequence of actions at the leaves of T, a_c be the last executed action, and s_c be the current state (see Figure 1), which will equal $\gamma(s_0, \pi[\preceq a_c])$ unless a disruption occurs. Then the executed and unexecuted parts of T are $T_x = T[\preceq a_c]$ and $T_u = T[\succ a_c]$, and the executed and unexecuted parts of π are $\pi_x = \pi[\preceq a_c] = \mathrm{plan}(T_x)$ and $\pi_u = \pi[\succ a_c] = \mathrm{plan}(T_u)$. A task t in T is *fully executed* if $T[t]$ is a subtree of $T[\preceq a_c]$, *unexecuted* if $T[t]$ is a subtree of $T[\succ a_c]$, and *partially executed* otherwise.

3 HTN Plan Repair Algorithms

Differences among the three algorithms color the experimental results and are critical to their interpretation. Briefly:

- Because REWRITE (or RW for short) always respects π_x, it considers unsolvable some of the problems that IPYHOPPER (IPH) or SHOPFIXER (SF) solve.
- SF detects potential failures in T_u using causal-link analysis; while costly, this means failures can be more rapidly found and repaired.
- IPH predicts the future via simulation, which incurs no additional cost unless a disturbance actually occurs.

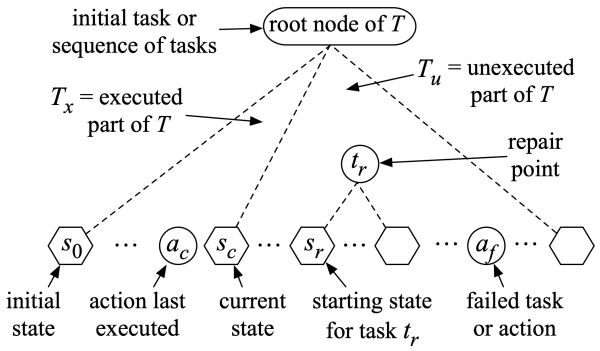

Figure 1: States and tasks relevant for repair of T.

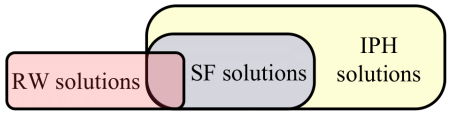

Figure 2: Venn diagram displaying the relationships among the sets of solutions each algorithm can produce. Note that $\{\text{RW solutions}\} \cap \{\text{IPH solutions}\} \subseteq \{\text{SF solutions}\}$.

- SF performs backjumping in T while IPH performs chronological backtracking, while again more costly for SF, it can resolve repairs in a more holistic manner.
- Section 4 proves the solution sets are as in Figure 2: all overlap, but RW and IPH both find some unique solutions and $\{\text{SF solutions}\} \subseteq \{\text{IPH solutions}\}$.
- To repair a plan, RW wants an exact match to the original plan's action prefix. So long as they are valid decompositions, there is no restriction on the methods used to produce said prefix. The rest of the plan is only restricted by decomposition validity without regard to plan stability.
- To try to preserve plan stability, IPH and SF try to restrict their changes to as small a part of the plan tree as possible. Unlike RW, they will retry failed tasks *ab initio*. For example, suppose task T is initially expanded to a, b, c, and after b is executed c's preconditions are unsatisfied. If another method is available that expands T to a', b', c', IPH and SF will try this method, giving a repaired plan of $a, b, a', b'c'$. RW would reject this plan, since the plan library cannot generate $a, b, a', b'c'$ from T.

REWRITE (RW): Compile a New Problem

The *rewrite* method (Höller et al. 2020b) compiles a new problem and domain definition that is solvable iff the plan can be repaired. This forces the planner to build a new plan with a prefix identical to π_x. As a compilation approach, RW is agnostic to the HTN planner or the search method used. However, there are limits to this flexibility: in practice HTN planners are far less uniform than PDDL planners.

The new compilation creates a new HTN problem by combining Σ, π (but not T), s_c, and the disturbance for a_c. The action a_c is rewritten so that its effects include the disturbance resulting in s_c. A solution to the new problem is any decomposition plan that is consistent with π_x and is a

complete, grounded decomposition of the initial tasks in T.[1]

RW algorithm implementation No runnable implementation of Höller, *et al.*'s algorithm was available,[2] so we implemented it ourselves; we will share our implementation on GitHub under an open source license. RW can use any HDDL-compatible planner, but SHOP3 was the only planner we could find that could parse and solve the experimental problems.[3] Our implementation follows the original definition in using HDDL for its input and output formats. Our implementation *differs* from the original definition in being able to handle action and method *schemas*, rather than only handling ground actions and methods (*i.e.*, it is a *lifted* implementation). This required extensions to some parts of the original algorithm. Our implementation returns a lifted plan repair problem that can be used directly by the lifted HTN planner SHOP3 (Goldman and Kuter 2019), and that can be grounded for use with grounded planners.

SHOPFIXER (SF): Examine Causal Links

SF (Goldman, Kuter, and Freedman 2020) repairs plans generated by the forward-searching HTN planner, SHOP3. It constructs a graph of causal links and task decompositions to identify the minimal subset of T to repair. SF extends the notion of plan repair stability introduced by (Fox et al. 2006), and further develops their methods and experiments, which showed the advantages of plan repair over replanning.

When a disturbance is introduced into the plan at s_c, SF finds the node $d_f \in T$ whose preconditions are violated; this will either be an action or a method. If there is no d_f, repair is unnecessary. Otherwise, SF "freezes" $\pi[\preceq a_c]$ and restarts SHOP3 from the parent of d_f's immediate parent.

Repair proceeds by *backjumping* into the search stack and reconstructing the compromised subtree without the later tasks. SF might backjump to decisions prior to d_f, but it cannot undo $\pi[\preceq a_c]$. SF returns T_x, $\pi[\preceq a_c]$, and a newly repaired suffix.

Furthermore, if p is the parent of child c in an HTN plan, then p's preconditions are considered chronologically prior to c's, because it is the satisfaction of p's preconditions that enables c to be introduced into the plan: if both p and c fail, and we repair only c, we will still have a failed plan, because after the disturbance, we are not licensed to insert c or its successor nodes.

[1]Their HDDL (Höller et al. 2020a) input notation permits problems that have both initial task networks *and* goals.

[2]Daniel Höller, personal communication, 26 September 2023.

[3]We tried Panda, which was only able to parse the smallest domains in 5 minutes (e.g., only Rovers 1-7). Panda's parsing and grounding algorithms explode in the presence of conditional effects and quantification. We confirmed this limitation with one of the Panda developers. Since preparing this manuscript, we have also checked with the IPC2023 HDDL planners Aries (Bit-Monnot 2023) and SIADEX/HPDL (Fernandez-Olivares, Vellido, and Castillo 2021). Aries cannot parse our domains, and while SIADEX appears to successfully parse our domains, it is unable to solve them.

IPYHOPPER (IPH): Repair via Simulation

IPH (Zaidins, Roberts, and Nau 2023) is an augmentation of IPyHOP, a progression-based planner (Bansod et al. 2022). In contrast to other HTN planners, it does not use projection and instead uses an external simulator to predict action effects.

For plan repair, IPH restarts the planning process at the task that was decomposed to produce the failed action using the current state in place of the stored state. Initially, IPH restricts the process to this subtree and only backtracks further up the tree when all decompositions in the subtree fail. Once it finds a valid decomposition, IPH simulates the action execution going forward. If simulation completes, the repair is considered successful. If simulation fails at some action a_f, IPH restarts a repair process at a_f. This simulation-repair cycle continues until either the plan successfully repaired or root is reached, indicating that no repaired plan is possible.

4 Theoretical Results

Preliminary Definitions

We begin by defining applicability of a decomposition tree T. This is more restrictive than just the applicability of $\mathrm{plan}(T)$. Not only must the actions' preconditions be satisfied, but also the preconditions of their ancestor methods. At each node of T, the preconditions that must be satisfied are given by a function $\mathrm{pre}^*()$ that is defined recursively:[4]

- If a ground method m in T is the child of a task t, then $\mathrm{pre}^*(m) = \mathrm{pre}(m) \cup \mathrm{pre}^*(t)$.
- If a nonprimitive task t in T is the first child of a ground method m, then $\mathrm{pre}^*(t) = \mathrm{pre}^*(m)$. Otherwise $\mathrm{pre}^*(t) = \varnothing$.
- If a primitive task (i.e., an action) a in T is the first child of a ground method m, then $\mathrm{pre}^*(t) = \mathrm{pre}(a) \cup \mathrm{pre}^*(m)$. Otherwise $\mathrm{pre}^*(t) = \mathrm{pre}(a)$.

We can now define applicability of T in a state s. Let $\pi = \mathrm{plan}(T)$, and suppose that for every action $a \in \pi$, the state $\gamma(s, \pi[\prec a])$ satisfies $\mathrm{pre}^*(a)$. Then T is applicable in s, and $\gamma(s, T) = \gamma(s, \pi)$. Otherwise, T is not applicable in s, and $\gamma(s, T)$ is undefined.

Finally, we define a notation for modifications to T: $\mathrm{replace}(T, \langle t_1, \ldots, t_n \rangle, \langle T'_1, \ldots, T'_n \rangle)$ is a modified version of T in which $T[t_i]$ is replaced with T'_i for $i = 1, \ldots, n$. As a special case, $\mathrm{replace}(T, t, T') = \mathrm{replace}(T, \langle t \rangle, \langle T' \rangle)$.

Plan Repair Definitions

Let a_c be the last executed action, and s_c be the observed current state. We consider four (non-disjoint) classes of HTN plan-repair problems:

Class 1: normal execution. If $s_c = \gamma(s_0, \pi[\preceq a_c])$, then s_c is the predicted state, so no repair is needed.

[4]Formally, pre^* is a function of a node n, but we'll simplify the presentation by writing it as a function of the method or task that labels n. We hope the intended meaning is clear.

Class 2: anomaly repair. If $s_c \neq \gamma(s_0, \pi[\preceq a_c])$, then s_c is *anomalous*. To repair this condition, the *repair-by-rewrite* definition involves rewriting the planning domain so that a_c produces the state s_c instead of the state $\gamma(s_0, T[\prec a_c])$ (for details, see (Höller et al. 2020b)). Let $\overline{\Sigma}$ be the rewritten domain, and $\overline{\tau}$ be the rewritten version of τ in $\overline{\Sigma}$. Let $\overline{T}$ be any solution tree for the rewritten planning problem $(\overline{\Sigma}, s_0, \overline{\tau})$, and $\overline{T}_u$ be the unexecuted part of $\overline{T}$. Let T'_u be the corresponding decomposition tree in Σ (i.e., with every rewritten action in $\overline{T}_u$ replaced with its un-rewritten equivalent). Then T'_u is a repair of T_u.

Class 2 is the set of plan-repair problems that are defined in (Höller et al. 2020b) and are implemented by the RW algorithm. RW requires replanning whenever an anomaly occurs—but unless there is also a predicted task failure (see Class 3), T'_u may be identical to T_u.

Class 3: predicted-task-failure repair. In addition to s_c being anomalous, suppose T_u isn't applicable in s_c. Then there is a task t_f in T_u such that $\gamma(s_c, T_u[\prec t_f]) \not\models \mathrm{pre}^*(t_f)$. This is a *predicted task failure*, which may be repaired as follows. The *repair point* may be any task t_r in $T_u[\preceq a_f]$. Let s_r be the state immediately before $T[t_r]$. Note that in some cases, s_r may precede s_c.

If t_r is unexecuted, let T'_r be any solution tree for the planning problem $(\Sigma, s_r, \langle t_r \rangle)$, and let $T'_u = \mathrm{replace}(T_u, t_r, T'_r)$. But if t_r is partially executed, let $\langle t_1, \ldots, t_n \rangle$ be the postorder sequence of all unexecuted tasks in $T[t_r]$ that have no unexecuted ancestors, T'_r be any solution tree for the planning problem $(\Sigma, s_c, \langle t_1, \ldots, t_n \rangle)$, and $T'_u = \mathrm{replace}(T_u, \langle t_1, \ldots, t_n \rangle, \langle T'_r[t_1], \ldots, T'_r[t_n] \rangle)$. Then a repair of T_u may defined inductively as follows:

- (base case) if T'_u is applicable in s_c, it is a repair of T_u.
- (induction step) otherwise, T'_u must contain an action a'_f such that $\gamma(s_c, T'_u[\prec a'_f])$ doesn't satisfy $\mathrm{pre}^*(a'_f)$, i.e., a'_f is another predicted failure. In this case, if T''_u is any repair of T'_u, then it is also a repair of T_u.

Class 4: predicted-action-failure repair. In addition to the anomaly and predicted task failure, suppose $\gamma(s_c, \pi_u[\prec a_f]) \not\models \mathrm{pre}(a_f)$. This is a *predicted action failure*. The definition of a repair for this condition is the same as in Class 3, except that only the *actions* of T'_u are considered in the repair. More specifically:

- The base case applies if $\mathrm{plan}(T'_u)$ is applicable in s_c, and otherwise the induction step applies.
- The induction step uses the preconditions of the failed action (i.e., $\mathrm{pre}(a'_f)$) instead of the preconditions of the tree rooted above the failed action (i.e., $\mathrm{pre}^*(a'_f)$).

Theorems

The theorems in this section establish properties of IPH, SF, and RW, assuming that they each are called at the same point in the execution of the same plan. In particular, the theorems establish relationships among these algorithms and the Class 2, 3, and 4 solutions to HTN plan repair problems. We will use the following terminology:

- Since IPH and SF are nondeterministic, each defines a set of possible solutions to a repair problem. We will call these the *IPH solutions* and *SF solutions*, respectively.
- As discussed earlier, Class 2 repair problems and their solutions are essentially the same as in (Höller et al. 2020b). We will call the solutions the *RW solutions*.

Theorem 1 *In every HTN plan-repair problem, the set of Class 3 solutions is the set of SF solutions.*

Proof For the base case of Class 3, T'_u is applicable in s_c. In Algorithm S1, SF, such a T'_u will be returned as it meets the criteria for the second branch of the If statement. The base case of SF mimics the base case for Class 3. For the induction step, T'_u is not applicable in s_c. Therefore there exists at least one action in T'_u, a'_f such that $\gamma(s_c, \mathrm{plan}(T'_u[\prec a'_f])) \not\models \mathrm{pre}^*(a'_f)$. The third branch of the If statement will handle this and find the first such action, a'_f. It will then make a recursive call using T'_u and a'_f, nondeterministically find an ancestor task t_r of a'_f, and produce a repaired subtree T_r rooted at t_r. The unexecuted solution tree will then have the repaired subtree inserted as in the replace protocol described. This will continue until either plan repair fails or the base case is reached. Thus nondeterministic SF mimics the induction step of Class 3. As the base case and induction step of Class 3 are mimicked by SF, the set of all possible outputs of SF when there is a possible repair is equivalent to Class 3. If no Class 3 repair exists, the algorithm will exit at line 5 without returning any solutions, hence the set of Class 3 solutions is again the set of solutions returned by SF. □

Theorem 2 *In every HTN plan-repair problem, the set of Class 4 solutions is the set of IPH solutions.*

Proof Class 4 is the same as Class 3, except that:

- The base case applies if $\mathrm{plan}(T'_u)$ is applicable in s_c, and otherwise the induction step applies.
- In the induction step, $\mathrm{pre}^*(a'_f)$ is replaced with $\mathrm{pre}(a'_f)$.

Algorithm 1 is the same as Algorithm S1, except for the above as well. As Class 3 is to Class 4, so too is SF to IPH. As Class 3 is equivalent to the set of solutions producible by SF, Class 4 is equivalent to the set of solutions producible by SF. □

The remaining theorems in this section establish relationships among the sets of RW, SF, and IPH solutions. The Venn diagram in Figure 2 illustrates these relationships.

For Theorems 3 and 4, our proofs use the example shown in Figure 3. The upper tree is the original solution tree, the lower left is the tree that is a RW tree and not a SF tree, and the lower right is the SF tree that is not a RW tree. a_1 is executed, but an anomalous state, s_3, is observed rather than the expected state s_1. The state to each action's immediate left is the action's input state, with s_2 being the predicted ending state.

Theorem 3 *There are HTN plan-repair problems that have RW solutions but no SF solutions.*

Proof In the bottom left of Figure 3, we show a RW solution tree that is not a SF solution. The tree is applicable in the initial state, s_0, but not in the current state s_3. SF would

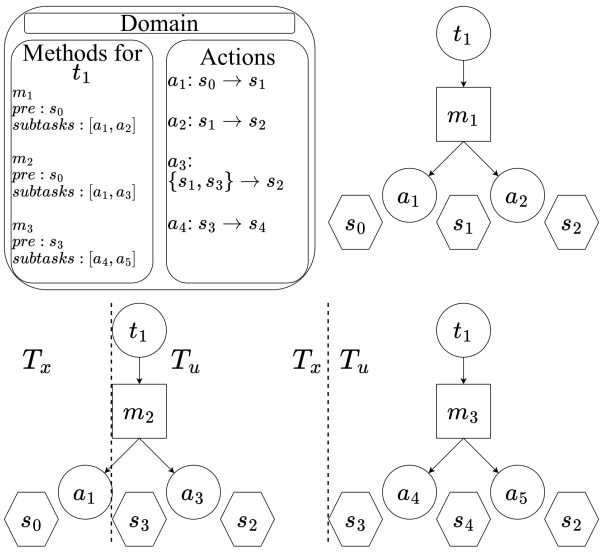

Figure 3: Example showing that there are problems that RW can solve but SF and IPH cannot, and *vice versa*.

require the parent task of a_3, t_1, to be decomposed with respect to the observed state, s_3. RW allows the methods of executed action be arbitrarily modified so long as the executed action prefix is preserved. With only m_1 and m_2 in the domain, the problem would have a RW solution, but not a SF solution. Thus there exist HTN plan-repair problems that have RW solutions but no SF solutions. □

Theorem 4 *There are HTN plan-repair problems that have SF solutions but no RW solutions.*

Proof In the bottom right of Figure 3, we show a solution tree that SF can produce, but not RW. The tree is applicable in the current state, s_3, but not in the initial state s_0. RW requires the preservation of the executed action prefix in the new solution tree, which is violated here. With only m_1 and m_3 in the domain, the problem would have a SF solution, but not a RW solution. Thus there exist HTN plan-repair problems that have SF solutions but no RW solutions. □

Theorem 5 *For every HTN plan-repair problem, every SF solution is also an IPH solution.*

Proof Recall that the SF solutions and IPH solutions are the Class 3 and Class 4 solutions, respectively. If T'_u is applicable in s_c, then $\mathrm{plan}(T'_u)$ is applicable in s_c. If $\mathrm{plan}(T'_u)$ is not applicable in s_c, T'_u is not applicable in s_c. If $s_c \models \mathrm{pre}^*(a'_f)$, then $s_c \models \mathrm{pre}(a'_f)$ as $\mathrm{pre}^*(a'_f) \models \mathrm{pre}(a'_f)$. Thus for each change we made to Class 3 to describe Class 4 we have loosened the constraints without adding constraints. Every Class 4 solution meets the criteria of Class 3 solutions, but not always the reverse. Thus the set of SF solutions is a subset (and often a proper subset) of the IPH solutions. □

Corollary 1 *There are HTN plan-repair problems that have IPH solutions but no RW solutions.*

Proof Immediate from the two preceding theorems. □

Theorem 6 *In every HTN plan-repair problem, if a repair is both a RW solution and IPH solution, it is also a SF solution.*

Proof If T'_u is a RW solution, $\overline{T}$ must be applicable in s_0. $\overline{T}_u[\succ a_c]$ is the same as T'_u, as $\overline{\Sigma}$ is the same as Σ beyond the executed action prefix. If T'_u is achievable by IPH, $\mathrm{plan}(T'_u)$ is applicable in s_c. T'_u is a subtree of $\overline{T}$. Every subtree of an applicable solution tree must be applicable. If $\mathrm{plan}(T'_u)$ is applicable in s_c and T'_u is a subtree of $\overline{T}$, T'_u is applicable in s_c. This satisfies the SF criterion of repair and therefore any solution that is both a RW solution and an IPH solution must also be a SF solution. □

Algorithm 1: Nondeterministic IPH pseudocode

```
1  Def IPH current execution state: s_c, unexecuted solution
     tree: T_u, failed action: a_f:
2      t_r, T'_r ← nondeterministically choose from set of
         tuples {t_a, T_a — t_a is an ancestor of a_f,
         decomposition tree T_a is rooted at t_a and applicable
         in γ(s_c, T_u[≺ a_f]} ;
3      T'_u ← replace(T_u, t_r, T'_r);
4      if T'_r is None then
5          return False ;
6      else if plan(T'_u) is applicable in s_c then
7          return T'_u ;
8      else
9          a'_f ← first action, a ∈ T'_u, where
             γ(s_c, T'_u[≺ a]) ⊭ pre(a) ;
10         return IPH s_c, T'_u, a'_f ;
```

5 Experimental Design

We tested SF, IPH, and RW on a set of identical initial plans and disturbances from three domains: *Rovers*, *Satellite*, and *Openstacks*. These are all HTN domains, formalized equivalently in HDDL and in SHOP3's input language. All of these domains were adapted from International Planning Competition (IPC) PDDL domains predating the HTN track, with HTN methods added and PDDL goals translated into tasks. These domains (with slightly different disturbances) were used in a previously published evaluation of the SF plan repair method (Goldman, Kuter, and Freedman 2020).

The Satellite and Rover domains each have 20 problems, and Openstacks has 30. For each domain, we ran 50 batches, where each batch was a run of each problem with one injected disturbance, randomly chosen and randomly placed in the original plan. All original plans were generated by SHOP3; they were translated into HDDL for IPH and RW. For IPH, all inputs were translated into JSON to avoid the need for a new HDDL parser. Runtimes were measured to the nearest hundredth of a second. All runtimes were wall-clock times, not CPU times. Runs were forcibly terminated after 300s and in such cases were marked as having failed. A runtime is successful when a valid repaired plan is returned.

Planning Domains

Here we briefly introduce the domains and deviation operators used in our experiments. All three domains used the

PDDL/HDDL ADL dialect. Domains were modeled equivalently in both HDDL and SHOP3; we have indicated below where the SHOP3 and HDDL domains diverged. All the *original* plans, that is, the plans to repair when execution-time deviations occur, were generated by SHOP3.

Deviations were modeled similarly to actions, with preconditions. Deviation preconditions and effects were defined in ways that aimed to avoid making repair problems unsolvable, but we were unable to make them completely safe. Non-trivial plan disturbances were difficult to model without rendering problems unsolvable because the limited expressive power of PDDL (and by extension HDDL) forced ramifications to be "compiled into" action effects (*e.g.*, counting the number of open stacks in the Openstacks domain), introducing dependencies that often could not be undone (*e.g.* failing the "send" operation in Openstacks had to also restore the relevant order to "waiting").

Rovers The Rovers domain is taken from the third IPC in 2002. Long & Fox say it is "motivated by the 2003 Mars Exploration Rover (MER) missions and the planned 2009 Mars Science Laboratory (MSL) mission." (Long and Fox 2003) The objective is to use a set of mobile rovers to go between waypoints, carrying out data-collection missions and transmitting data back to a lander. There are constraints on the lander's visibility from various locations, and on the rovers' ability to go between particular pairs of waypoints. Rovers problems scale in terms of size of the map, number of goals, and the number of rovers. Disturbances applied include losing collected data; decalibration of cameras; and loss of visibility between points on the map. For the Rovers problem, the SHOP3 domain uses a small set of path-finding axioms to guide navigation between waypoints. To avoid infinite loops in the navigation search space, IPH does not use lookahead in the waypoint map, but does detect and reject cycles in the state space (this check is sound but not complete).

Satellite The Satellite problem also premiered in 2002, and is described as "inspired by the problem of scheduling satellite observations. The problems involve satellites collecting and storing data using different instruments to observe a selection of targets." (Long and Fox 2003) Disturbances used were changes in direction of satellites, decalibration, and power loss. Problems scale by number of instruments, satellites and image acquisition goals.

Openstacks IPC 2006 introduced the Openstacks domains as a translation of a standard combinatorial optimization problem, "minimum maximum open stacks," in which a manufacturer needed to fill a number of orders, each for a combination of different products (Gerevini et al. 2009). Problems scale by numbers of products, number of orders, and number of products in an order.

For plan repair, deviations included removing previously-made products, and shipping-operation failures. These were difficult to add to Openstacks without introducing dead ends into the search space because of consistency constraints on the state that are only implicit in the operators. Thus to make repair possible, we added a "reset" operation to reset an order from "started" to "waiting."

The SHOP3 domain for Openstacks included axioms for a cost heuristic. This is a common difficulty in plan repair: typically domains are not written in such a way that recovery is possible if a plan encounters disturbances because limitations in expressive power mean ramifications must be programmed into the operator definitions. Furthermore, since state constraints (*e.g.*, graph connectivity in the logistics domain) cannot be captured in PDDL, one can inadvertently make dramatic changes to problem structure by introducing disturbances; cf. Hoffmann (2011) on problem topology.

6 Results

Satellite The Satellite domain was the easiest for all three repair methods. Both IPH and SF solved all of the repair problems in our data set, and found solutions quickly. RW solved the majority of the problems, between 60% and 85% of them (Figure S1). Inspection shows that it correctly solved all the problems that did not need repair (*i.e.*, it never timed out re-deriving a plan). Figure S2 gives the runtimes for all three methods, using $\log_{10}$ because of the wide range of values. IPH runtimes are slightly better than SF on average. The times for RW are almost uniformly worse, and scale worse as the problem size grows. The results for RW are not surprising, since proving unsolvability may take longer. Indeed, plotting the runtimes for success and failure separately, demonstrates that (Figure S3). Interestingly, there are no failures due to timeout: RW is able to prove unsolvable all of the unrepairable problems). While the IPH times are generally the best on average, its times vary more widely: see Figure S4. Standard deviation of run times is greater for IPH than SF for all except problem 20.

Rovers The Rovers repair problems were more difficult than Satellite, and none of the repair methods solved all of them. Figure 4 shows success percentages. Note the outliers for IPH and SF in problems 3 and 6 (which are also difficult for RW) and for IPH in problems 10 and 20. Generally, SF is more successful than IPH, as shown in Table 1.

Generally, while RW was less successful than the other algorithms, it tracks their results except for problems 14 and 15, where the other two are uniformly successful, but RW is only 86% and 92% successful.

Runtimes are graphed in Figures 5, and 6. We plot the successful and failed runs separately, because the failed runs include both cases where an algorithm proves that the problem is unsolvable and cases where it simply runs out of time (time limit was set at 300s).

Again, IPH is generally faster, but SF scales better with problem difficulty. For IPH, problem 3 is an outlier in elapsed time due to failures, SF has issues with problem 5, and all three have difficulties with 6 due to failures. RW has great difficulty with the larger problems. On the successful problems, IPH has a higher runtime variance than SF: see Figure S8 in supplemental for details. For all 20 problems, IPH's standard deviation of runtime is greater than SF's.

Openstacks For Openstacks, both SF and RW solved all the repair problems. IPH solved *almost* all, but failed for a small number (see Table 2). The runtimes, graphed in Figure S9 clearly show that RW and IPH do not scale well on

Figure 4: Success rates for the Rovers repair problems for each of the three algorithms.

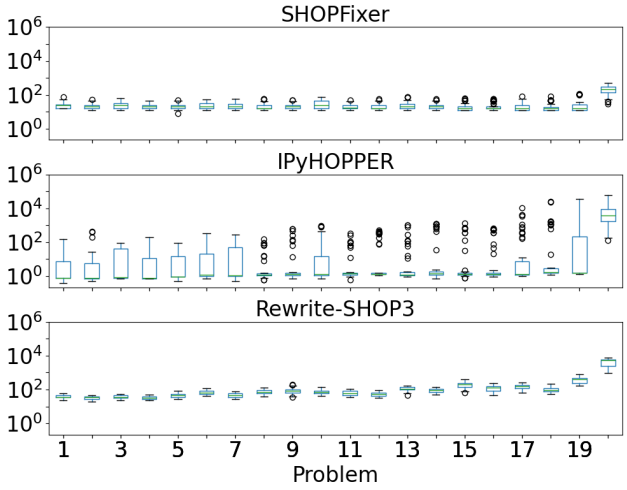

Figure 5: Each algorithm's runtimes in msec (semi-log plot) on the Rovers repair problems that the algorithm solved successfully.

the more difficult problems, with RW notably worse. IPH runtimes for problems 3 and 6 are outliers: they are much more difficult than for this solver than for the others. Full details are given in Table S2.

7 Discussion

Common Features Across all of the domains, RW is less time-efficient than the other two repair methods. This is due to the fact that it replans *ab initio*, albeit against a new problem that forces the plan to replicate already-executed actions. This involves an extensive amount of rework, as can be seen very clearly in the Openstacks problems, which have the highest runtimes for generating initial plans. This could likely be substantially improved by heuristic guidance to direct the early part of planning towards methods that replicate actions previously seen and that avoid infeasibly introducing new actions.

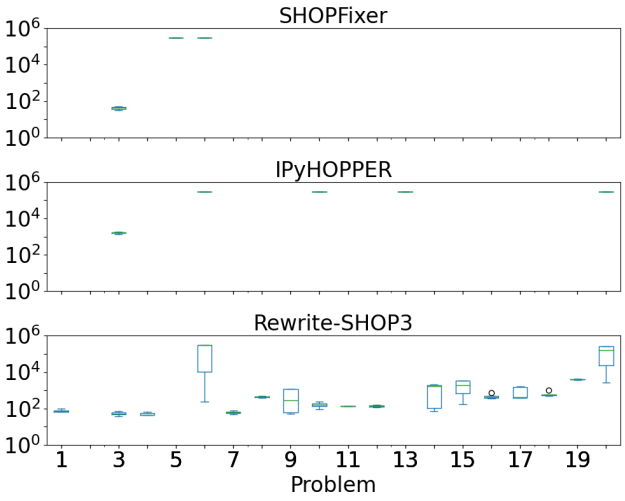

Figure 6: Each algorithm's runtimes in msec (semi-log plot) on the Rovers repair problems that the algorithm did *not* solve successfully.

Problem	IPH % success	SF % success
3	64%	**72%**
5	**100%**	94%
6	**80%**	**80%**
10	74%	**100%**
13	96%	**100%**
20	36%	**100%**

Table 1: Success rates for Rover problems where either IPH or SF did not solve all repair problems. Problems not listed were all solved by both repair methods.

Satellite It is unsurprising that RW cannot solve some problems that the other two algorithms solve. Many of the repairs for satellite simply involve immediately restoring a condition deleted by a disturbance—and if it was a condition that the plan established, a re-establishment typically will not work. Perhaps the surprise is that *any* of these scenarios are successfully repaired by RW. We investigated further, and found that RW could handle all of the cases that did not require repair (where the disturbance did not defeat any action preconditions). Removing those cases gives the success rates shown in Figure S1.

Here is an example of how RW's definition of repair makes it unable to solve a problem handled by the other two systems. In this repair problem , `instrument0` becomes decalibrated after it has been calibrated and pointed at its observation target (`phenomenon6`). The way the domain is written, calibration occurs only in a sequence of calibration then observation. Thus there is no plan in which two calibration operations are not separated by an observation, so repair by rewrite is impossible. The other two methods simply treat the preparation task as having failed, and re-execute the calibration, because their repair definition is more permissive.

For this domain, IPH is generally faster than SF, although it is implemented in interpreted Python rather than compiled

Problem	% success	Problem	% success
7	98%	27	96%
10	98%	28	72%
14	98%	29	78%
19	98%	30	78%
25	94%		

Table 2: Success rates for Openstacks problems where IPyHOPPER did not solve all of the problems.

Common Lisp. This is probably because SF invests in building complex plan trees that include dependency information in order to more rapidly identify the location For these simple problems, SF's added effort is generally not worthwhile. We note that the *variance* of IPH's runtimes is wider than that of SF, and that there are more outliers (Figure S4).

Rovers There were several Rovers problems where even IPH and SF could not find solutions—but the three algorithms behaved quite differently in these cases. There were 99 Rovers problems that Rewrite-SHOP3 could not repair. Of these, only 2 were due to timeouts, both on problem 6, showing that the algorithm usually could prove problems unrepairable. As before, SF's more permissive definition of "repairable" meant that it solved more problems: it found only 35 unrepairable, and of these only 3 were due to timeouts, as with Rewrite-SHOP3 these were both for problem 6. IPH had much more difficulty with these problems. It failed to repair 97 cases, of which 78 were due to timeouts. Timeouts are not a simple matter of scale: the greatest number of timeouts (by a factor of 2) is for problem 20, but the runners-up are 3, 6, and 10, in declining order of number of timeouts. Since the problems are intended to scale from first to last, the outcomes are not due only to raw scale.

Repair difficulties in the Rovers domain are due to the nature of the disturbances in our model. The "obstruct-visibility" disturbance can render the waypoint graph no longer fully connected, in terms of rover reachability. Losing a sample may also give rise to an unrepairable problem.

Openstacks The hardest of the domains, Openstacks shows the benefit of SF's more expensive tree representation in runtime. We can see this even more clearly by plotting the two algorithms against each other (Figure S12). IPH's failures in this domain are all due to timeouts. Specific problems are indicated in Table 2. The more difficult search space here heavily penalizes IPH's simple chronological backtracking.

In a reversal of the previous patterns, RW solves all of the problems. This is due to a difference in the way the domain was formalized. Recall that we had to add a new action to prevent disturbances from making the Openstacks problems unsolvable. That modification had the effect of also helping RW as did the fact that action choice is primarily constrained by preconditions, rather than by method structure, which also avoided issues with this algorithm. Note that this relatively unconstrained planning also made it more difficult to generate the initial plans for this domain.

8 Conclusions and Future Directions

We have presented an analysis of three recent hierarchical repair algorithms from the AI planning literature: SF, IPH, and RW. Qualitatively, our analyses highlighted significant differences among these methods: First, RW's definition of plan repair is more stringent than the others in practice; we have identified benchmark planning problems that are solvable for IPH and/or SF that cannot be solved by RW. Secondly, SF attempts to detect when a plan will be invalid, before any actions actually fail; i.e., SF invests in both data structures and computation to detect compromise to a plan as soon as possible. In contrast, IPH is not a model-based projective planner in the same sense: it relies on an external simulation to do projection for it, instead of having an internal action model as most planners do. This difference in planning approaches leads to different plan repair behaviors.

Our results on the efficiency of the RW algorithm should be taken with a large grain of salt. The original developers of this algorithm point out that their characterization is intended to be conceptually correct and clean, and that they have not yet taken into account the efficiency of the formulation. In addition to tuning the formulation, its efficiency could be improved by improved heuristics for planner when they run against rewrite problems. In particular, a planner searching the decomposition tree top-down should take into account the position of its leftmost child when deciding whether to choose the original methods, or methods whose leaves are taken from the executed prefix of the plan.

Our experiences also highlight unresolved issues in applying RW in grounded planning systems. How best to schedule re-grounding *vis-à-vis* generation of the rewritten repair problem remains to be determined. While there were some subtleties to resolve in developing our lifted implementation, it did not have this chicken-and-egg problem.

Another interesting research direction is to study how HTN domain engineering affects the tradeoffs between efficiency and flexibility. At present, repair problems are generally created by modifying previously-existing planning problems (including IPC problems) that were not designed for execution, let alone to be repairable. In connection with the general concept of stability, this may yield a new insights in search-control for plan repair and for repairable plans; deriving properties from how preconditions and effects enable planning heuristics and repairability as well as how those preconditions and the task structure enable search control at higher levels of the plan trees. Like *refineability* properties (Bacchus and Yang 1992; Yang 1997), this approach may be examined formally, theoretically, and experimentally.

Acknowledgements

This project is sponsored by the Air Force Research Laboratory (AFRL) under contract FA8750- 23-C-0515 for the HI-DE-HO STTR Phase 2 program. Distribution Statement A. Approved for public release: distribution is unlimited. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFRL. Thanks to the anonymous reviewers for their helpful feedback. MR thanks ONR and NRL for funding portions of his research.

References

Ayan, F.; Kuter, U.; Yaman, F.; and Goldman, R. P. 2007. HOTRiDE: Hierarchical Ordered Task Replanning in Dynamic Environments. In *ICAPS-07 Workshop PlanEx*.

Bacchus, F.; and Yang, Q. 1992. The expected value of hierarchical problem-solving. In *AAAI*, 369–374. Citeseer.

Bansod, Y.; Patra, S.; Nau, D.; and Roberts, M. 2022. HTN Replanning from the Middle. *The International FLAIRS Conference Proceedings*, 35.

Bercher, P.; Biundo, S.; Geier, T.; Hoernle, T.; Nothdurft, F.; Richter, F.; and Schattenberg, B. 2014. Plan, Repair, Execute, Explain — How Planning Helps to Assemble Your Home Theater. In *Proceedings of ICAPS*, volume 24, 386–394.

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

Fernandez-Olivares, J.; Vellido, I.; and Castillo, L. 2021. Addressing HTN Planning with Blind Depth First Search. *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts–Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, 1–4.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan Stability: Replanning versus Plan Repair. In *ICAPS*.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic Planning in the Fifth IPC: PDDL3 and Experimental Evaluation of the Planners. *AI*, 173(5-6): 619–668.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Amsterdam Boston: Elsevier/Morgan Kaufmann.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: The Case of SHOP3. In *Proceedings of the 12th European Lisp Symposium*. Genova, Italy.

Goldman, R. P.; Kuter, U.; and Freedman, R. G. 2020. Stable Plan Repair for State-Space HTN Planning. In *HPlan 2020 Working Notes*. Nancy, France.

Hoffmann, J. 2011. Analyzing Search Topology without Running Any Search: On the Connection between Causal Graphs and h^*. *JAIR*, 41: 155–229.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020a. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *Proceedings of AAAI 2020)*. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN Plan Repair via Model Transformation. In Schmid, U.; Klügl, F.; and Wolter, D., eds., *KI 2020: Advances in Artificial Intelligence*, volume 12325 of *Lecture Notes in Computer Science*, 88–101. Cham: Springer International Publishing.

Kambhampati, S.; and Hendler, J. A. 1992. A Validation-Structure-Based Theory of Plan Modification and Reuse. *AIJ*, 55: 193–258.

Kuter, U. 2012. Dynamics of Behavior and Acting in Dynamic Environments: Forethought, Reaction, and Plan Repair. Technical Report 2012-1, SIFT.

Long, D.; and Fox, M. 2003. The 3rd IPC: Results and Analysis. *JAIR*, 20: 1–59.

Yang, Q. 1997. Generating Abstraction Hierarchies. *Intelligent Planning: A Decomposition and Abstraction Based Approach*, 189–206.

Zaidins, P.; Roberts, M.; and Nau, D. 2023. Implicit Dependency Detection for HTN Plan Repair. In *Proceedings of the ICAPS HPlan Workshop*. Prague, Czech Republic.