A Comparative Analysis of Plan Repair in HTN Planning

Robert P. Goldman[1], **Paul Zaidins**[2], **Ugur Kuter**[1], **Dana Nau**[2]

[1]SIFT, LLC
[2]University of Maryland, College Park

Abstract

This paper reports an analysis of three recent hierarchical plan repair algorithms: SHOPFIXER, IPYHOPPER, and REWRITE. We compare these algorithms qualitatively, and evaluate their performance, quantitatively, in a series of benchmark planning problems, informed by our qualitative analysis. A critical part of the qualitative comparison is that REWRITE, a problem-rewriting technique, has a substantially different and more restrictive *definition* of plan repair than the other two systems. Understanding this distinction will be important when choosing a repair method for any given application. Our results explain the runtime repair performance of these systems as well as the coverage of the repair problems solved, based on algorithmic properties such as chronological backtracking vs. backjumping over plan trees.

1 Introduction

Plan repair has been shown to provide advantages over generating new plans from scratch both in terms of planning runtime and plan stability – the amount of plan content that is retained between the original and repaired plans (Fox et al. 2006). Fox, *et al.* showed that plan repair could provide new plans faster, and with fewer revisions, than replanning *ab initio* in the face of disruptions. 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. The term "minimal perturbation" has been used synonymously (Cushing and Kambhampati 2005). To be precise, "stability" is actually a *relation*: a solver is stable if the size of the change in the output is proportional to the size of the change in its input: at least in theory, a minimal perturbation solver could actually be unstable. Plan stability is particularly important for human interaction, as users are confused by radical changes to plans introduced in response to trivial upsets.

The concept of stable plan repair has been generalized from classical planning to Hierarchical Task Network (HTN) planning. Early work on hierarchical plan repair introduced validation graphs in the context of hierarchical and partial-order causal link planning, where plan repair used validation graphs to identify disruptions and patches to the partial-order plans (Kambhampati and Hendler 1992). Extending classical plan repair on sequences of actions, hierarchical repair algorithms provide localization of errors and failures and problem refinement methods that take advantage of such localizations to provide better stability (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020).

Over the years, there have been great strides in HTN plan repair in which a variety of repair algorithms have been proposed. The three that we consider are SHOPFIXER (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020), IPYHOPPER (Zaidins, Roberts, and Nau 2023), and an unnamed algorithm that we will call REWRITE (Höller et al. 2020b). These build on several previous methods (Ayan et al. 2007; Kuter 2012; Bansod et al. 2022; Bercher et al. 2014).

We compare SHOPFIXER, IPYHOPPER, and REWRITE qualitatively, and evaluate their quantitative performance in a series of benchmark planning problems in the light of our qualitative analysis. Our results demonstrate the following:

- Because of differences in their notion of what repairs are permissible and how to go about doing them, there are differences in which repair problems REWRITE can solve as opposed to IPYHOPPER and SHOPFIXER, which share a definition. The three algorithms also differ in what kinds of repairs they make.
- The REWRITE repair method, which must replicate already-executed actions, involves extensive amount of re-derivation of plans, as can be seen in its worse runtimes for all of the domains.
- Chronological backtracking during hierarchical repair involves blindly trying a large number of subtrees of the original plan tree, most of which do not contribute to repairing the plan. In more complex problems, semantic (*i.e.*, causal) backjumping yields better performance, as can be seen in the Openstacks domain and the more difficult Rovers problems.
- Less-expensive simulation lookahead for repair provides a better payoff than extensive work in building data structures (*e.g.*, explicit causal links) to speed backtracking and backjumping in problems of modest scale, such as the Satellite domain.

An additional contribution of our work is to provide the first publicly-available implementation of the REWRITE repair algorithm. We also extended it to work in lifted domains, which is critical for completeness, since practical grounding methods typically include problem-specific pruning. Such pruning may compromise the completeness of the

plan repair method, since disturbances may render new parts of the state space reachable.

2 Hierarchical Plan Repair Strategies

The three algorithms analyzed in this paper have important qualitative differences that color the experimental results and are critical to their interpretation. First, REWRITE's definition of plan repair is more stringent than the others; we have found benchmark planning problems IPyHOPPER and/or SHOPFIXER solve but REWRITE does not. The second difference is between SHOPFIXER and IPyHOPPER. SHOPFIXER attempts to detect when a plan *will be* invalid, before any actions actually fail; it invests in data structures and computation in order to detect problems as soon as possible. IPyHOPPER's projections are not model-based: it relies on an external simulation to do projection for it, instead of having an internal action model as most planners do. These differences lead to different plan repair behaviors.

These differences are not simply a matter of one repair method being "better" than another: instead, different repair methods are better in different situations. The more stringent definition offered by REWRITE is better when an HTN method library captures important considerations about what sequences of actions are and are not correct plans; the more relaxed definition better when the precise trajectory is less important. SHOPFIXER's plan repair approach is better if the costs of wrong actions are higher than the costs of computation, *e.g.* when actions are particularly expensive, or when deferring repair could leave the agent trapped in a dead end. When a situation is more forgiving and when it changes frequently, SHOPFIXER's aggressive repair strategy will not be worthwhile. In the following, we give simple examples that illustrate these differences.

The REWRITE paper (Höller et al. 2020b) defines a repaired plan as one that, among other considerations, has a plan (decomposition) tree that is a refinement of the plan tree of the initial plan. This definition can exclude some repairs that seem intuitively plausible.

Consider an HTN plan domain for inter-city travel that has two alternative methods: rail and air travel. For rail travel, we take the bus to the station, embark, travel, and then debark. Similarly, to fly we take the bus to the airport, get on the plane, fly, and then deplane. Each method has the precondition that the train station (resp., airport) be open. Starting from home, Panda (Höller et al. 2021) and SHOP3 (Goldman and Kuter 2019) both can find plans for air and rail travel.

Consider the plan repair problem that occurs when we take the bus to the train station and discover that the train station is closed. SHOPFIXER and IPyHOPPER will detect the problem, identify that the original `rail-travel` method cannot be fixed, and switch to air travel. However, REWRITE cannot repair this plan, because the resulting plan – bus to train station, bus to airport, embark, fly, deplane – is ill-formed: no expansion of the top level goal contains *both* "bus to train station" *and* "bus to airport."

This highlights a tradeoff between flexibility and efficiency that the authors of HTN domain descriptions typically face. If the HTNs use strong search-control strategies and knowledge to make planning efficient by quickly finding a good and acceptable solution, then those HTNs are usually not flexible enough to allow exploring alternative plans for in hierarchical plan repair. As the above example shows, the REWRITE algorithm commits to the prefix of the hierarchy for planning reasons, but that excludes possible repairs at the higher levels of the hierarchy when a discrepancy occurs. SHOPFIXER and IPyHOPPER are similar in that limitation for general cases but they provide backtracking and backjumping strategies that can alleviate this limitation in some classes of domains. Section 4 discuss examples of this phenomenon in our experimental results.

The REWRITE paper gives an example that shows the rationale for its more restrictive definition of plan repair. In this example, the agent drives through a city that has congestion pricing, and must pay a toll for each road segment driven in the congestion zone. Solution plans have the form $x^*a^nb^nx^*$: there are actions before and after travel in the congestion zone (the two x^*'s), then n segments traveled in the congestion zone (a^n), followed by n toll payments (b^n). The structure of the HTN methods is the mechanism that enforces the a^n-b^n balance, so if extraneous actions were allowed, incorrect (unbalanced) plans could be derived.

SHOPFIXER and IPyHOPPER share the same model of plan repair, which holds that any HTN method may be redone – *i.e.*, its plan regenerated and then executed from the method's beginning, as long as the method's preconditions hold in the state in which it begins. As we have seen above, that means that the "intercity-travel" task may be restarted after the agent has reached the train station and discovered that it is closed. The difference between the two is how they attempt to detect future action failures.

When building a repairable plan, SHOPFIXER creates a decomposition tree that records task decompositions into subtasks, with cross-links from actions that establish facts to the actions and methods whose preconditions consume those facts. This trades some pre-computation and storage for more rapid detection of possible plan failures – and detection of cases where unexpected effects do not cause plan failures. An example SHOPFIXER tree is shown in Figure 1.

IPyHOPPER, in contrast, does not precompute any data structures for identifying plan failures. Instead, when notified of a plan disturbance, it simulates the existing plan forward in time to find impending failures. When it finds such a failure, it unexpands the simulated action's parent node in the solution tree, and attempts to find a new decomposition for that node that will not fail. If no such decomposition exists, the parent's parent is similarly unexpanded; this continues up the solution tree until either a valid decomposition is found or the root is reached, which means no repair exists.

If a valid decomposition is found, IPyHOPPER restarts the simulation at the leftmost action of the decomposition in postorder traversal. This continues until either the plan completes successfully or another failure is simulated. If another failure is simulated, the repair process is repeated and eventually either the simulation will complete and the plan is repaired or the root will be reached and repair has failed.

We now will briefly summarize the three hierarchical plan repair algorithms that we have evaluated in this work.

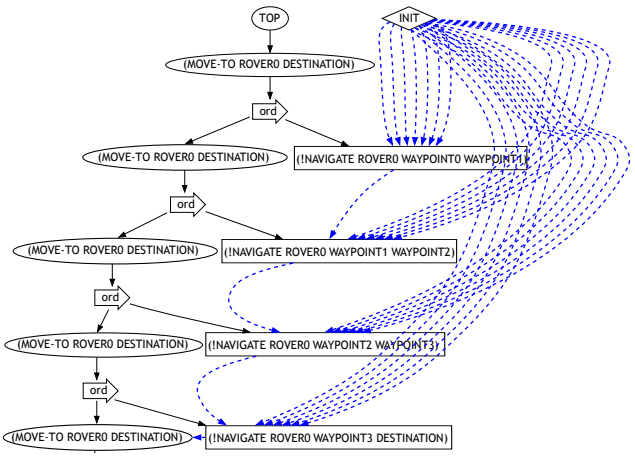

Figure 1: SHOP3 plan tree for a rover plan; decomposition edges are in black; dependency cross-edges in dashed blue. Edges from the diamond node represent dependencies on initial state facts.

SHOPFIXER Plan Repair

SHOPFIXER (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020) is a method for repairing plans generated by the forward-searching HTN planner, SHOP3. It uses a graph of causal links and task decompositions to identify a minimal subset of the plan that must be fixed. SHOPFIXER extends the notion of plan repair stability introduced by (Fox et al. 2006), and further develop their methods and experiments, which showed the advantages of plan repair over replanning.

The basic idea behind SHOPFIXER's plan repair approach is very simple: when a disturbance is introduced into the plan, SHOPFIXER finds the minimal subtree of the plan tree that contains the node whose preconditions are clobbered by that disturbance: the failure node. If there is no such node, then the disturbance does not interfere with the success of the plan. SHOPFIXER will then repair the plan, starting with the minimal subtree. To find the minimal subtree around a failure node, SHOPFIXER finds the first task in the plan that is potentially "clobbered" (rendered unexecutable) by that disturbance, and restarts the planning search from that task's immediate parent in the HTN plan (since that was where that task was chosen for insertion into the plan). This plan repair is done by *backjumping* into the search stack for SHOP3 and reconstructing the compromised subtree without the later tasks. Note that the first clobbered task may be either a primitive task or a complex task. 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.

SHOPFIXER restarts the planning search by backjumping to the corresponding entry in the SHOP3 search stack, which it retains, and updating the world state at that point with the effects of the disturbance. When restarting the planning search, SHOPFIXER "freezes" the prefix of the plan that has already been executed, as well as the deviation and its effects. It may backjump to decisions prior to the deviation, for example, if the immediate parent of the failed task is the top level task of the problem, but it cannot undo the effects of an action that is already done. SHOPFIXER returns a repaired plan that is made up of the prefix before the disturbance, the disturbance, and the repaired suffix.

IPyHOPPER Plan Repair

IPYHOPPER (Zaidins, Roberts, and Nau 2023) is a progression based HTN planner written in Python. The primary distinction between IPYHOPPER and other plan repair methods is that it does not rely on a projection algorithm. Instead, it uses an external simulator to predict the effects of planned actions.

IPYHOPPER's planning engine is an augmented version of the prior IPyHOP planner (Bansod et al. 2022). For planning, input is in the form of an initial task list, initial state, and domain description. The domain description includes tasks, primitive actions, and method definitions. The initial tasks are repeatedly decomposed into simpler tasks and then finally actions based on the domain description. The decomposition process forms a solution tree by a depth-first traversal and every intermediate state is saved in the tree for backtracking. When a task cannot be successfully decomposed, a new decomposition is attempted of the most recent task expanded. When every task has been decomposed and all actions' preconditions are established, planning has successfully completed and the actions of the tree in preorder constitute the plan. If the planner backtracks to the sentinel root node, which is the parent of all input tasks, planning has failed: no decomposition can achieve the task list in order.

For plan repair, IPYHOPPER restarts the planning process at the parent of the immediate parent of a failed action using the current state in place of the stored state. Initially, IPYHOPPER 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, we simulate the action execution going forward. If our simulation completes, the plan is repaired and the process is terminated. If IPYHOPPER encounters a future simulated failure, it will redo the repair process. This simulation-repair cycle continues until either the plan successfully repaired or root is reached, indicating that no repaired plan is possible.

Plan Repair by Problem Rewriting

The two methods we have discussed above both share the core pattern of resuming the planning process after some appropriate change to the search process. They generate a repaired plan by redoing some portion of planning process. The *rewrite* method of Höller, *et al.* (Höller et al. 2020b) is very different: it operates by generating a new problem and domain definition that is solvable iff the plan can be repaired. These definitions are generated by combining the original problem and domain definitions, the original solution (plan), the position reached in execution, and the disturbance. They

do not provide a method for determining whether a plan repair is actually required, so a repair problem must be solved after every disturbance. A key advantage of their algorithm is that it is not specific to any particular HTN planner: any planner that accepts their input format will work.

The central intuition behind the problem/domain method is to force the planner to build a new plan that has as a prefix the set of actions that were executed before the disturbance. The final action in this prefix is modified so that its effects include the disturbance effects. Any decomposition plan that is consistent with the observed actions and that contains an executable suffix that performs all the initial tasks, and achieves any specified goals.[1] This definition accounts for the distinction between repairs permitted by rewrite and those of SHOPFIXER and IPYHOPPER. The latter systems accept repaired plans that include methods that have been *abandoned* and their tasks achieved through new decompositions not consistent with the original plan: rewrite does not.

Rewrite 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. 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.

REWRITE generates a new planning domain and problem, so in theory it may be coupled with any HTN planner. In practice, since HTN problem definitions are less standardized than classical ones, there are limits to this flexibility. Our implementation returns a lifted plan repair problem that can be used directly by the lifted HTN planner SHOP3 SHOP3 (Goldman and Kuter 2019), and that can be grounded for use with grounded planners.

We had originally intended to report on experiments that used both SHOP3 and Panda (Höller et al. 2021) as planners for the repair problems. Unfortunately, we found that Panda was unable to handle the benchmark domains we have used in our experiments, namely, Rovers, Satellite, and OpenStacks as reported next section. These domains all use the ADL dialect of PDDL/HDDL, featuring quantified goals and conditional effects. The parsing and grounding methods used in Panda were not able to handle the demands of ADL domains (we confirmed this with Panda developers), so we had to limit ourselves to using only SHOP3.[3] We refer to this combination as Rewrite-SHOP3. However, since SHOP3 and Panda would be solving the same problems, we are still testing the essential features of the rewrite algorithm.

[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]None of the satellite problems could be parsed by Panda in 5 minutes, only 7 of the rovers, and 9 of the openstacks. The 7 rovers problems could be grounded, but only 7 of the 9 openstacks problems. Details available upon request.

Summary The REWRITE algorithm is most appropriate for problems where the structure of the plan tree is critical to correctness (e.g., the toll example, where movements and payments must be balanced by the tree), but it may fail where disturbances put the agent into a dead end that it will have to "back out" of. IPYHOPPER and SHOPFIXER will be the opposite: both allow for deviations in the plan tree, so both will easily handle cases that involve backing up to reverse a deviation and then resuming. They differ in that IPYHOPPER will more efficiently handle deviations with immediate impact, and SHOPFIXER can better handle deviations with delayed impact, at the expense of computation that will be wasted on simple cases.

3 Experimental Design

We tested SHOPFIXER, IPYHOPPER, and Rewrite-SHOP3 on a set of identical initial plans and disturbances from three domains: **rover**, **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 SHOPFIXER plan repair method (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020).

The Satellite and Rover domains each have 20 problems, and the Openstacks domain 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 IPYHOPPER and Rewrite-SHOP3. For IPYHOPPER, all inputs were translated into JSON to avoid the need for a new HDDL parser. We wish to emphasize that each repair method started with the same plan and same disturbance in each batch. So for each problem there are 50 disturbance examples, which we ran on all three of the algorithms. Runtimes were measured to the nearest hundredth of a second. All runtimes were wall-clock times, not CPU times: we were concerned that comparing Python CPU times with Common Lisp CPU times might not be valid. In the event, differences between CPU times and wall-clock times were negligible.

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*, to-be-repaired plans 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. 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. The objective is to use a collection of mobile rovers to traverse between waypoints on the planet, carrying out a variety of data-collection missions and transmitting data back to a lander. The problem includes constraints on the visibility of the lander from various locations and on the ability of individual rovers to traverse between particular pairs of waypoints." (Long and Fox 2003) 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, IPYHOPPER does not use lookahead in the waypoint map, but it does check for and reject cycles in the state space.

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 Openstacks was introduced, as a translation of a standard optimization problem, in IPC 2006:

> The Openstacks domains are ... based on the "minimum maximum open stacks" combinatorial optimisation problem ... A manufacturer has a number of orders, each for a combination of different products. Only one product can be made at a time, but the total required quantity of that product is made at that time. From the time that the first product requested by an order is made to the time that all products included in the order have been made, the order is said to be "open" and during this time it requires a "stack" .... The problem is to order the making of the different products so that the maximum number of stacks that are in use simultaneously ... is minimised. (Gerevini et al. 2009)

Problems scale by number of orders, number of products, and number of products in a single order. Deviations include removing products that were previously made and causing the shipping operation to fail. Deviations were particularly difficult to add to Openstacks without introducing dead ends into the search space, because there are consistency constraints on the state that are only implicit in the operators. Therefore, to make repair possible, we needed to add a "reset" operation that would reset an order to "waiting" mode from "started." The SHOP3 domain for Openstacks included axioms for a cost heuristic. Note that 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 means that ramifications must be programmed into the operator definitions. Furthermore, since state constraints (*e.g.*, graph connectivity in the logistics domain) are not and cannot be captured in PDDL, one can inadvertently make dramatic changes to problem structure by introducing disturbances; cf. Hoffmann (2011) on problem topology.

4 Results

Satellite The Satellite domain was the easiest for all three repair methods. Both IPYHOPPER and SHOPFIXER solved all of the repair problems in our data set, and solutions were found quickly. Rewrite-SHOP3 solved the majority of the problems, between 60% and 85% of them (Figure 2). Inspection showed that it correctly solved all the problems that did not need repair (*i.e.*, did not time out rederiving the plan). Figure 3 gives the runtimes for all three methods, using $\log_{10}$ because of the range of values. IPYHOPPER runtimes are slightly better than SHOPFIXER on average. The rewrite times are almost uniformly worse, and scale worse as the problem size grows. The results for Rewrite-SHOP3 are not surprising, since proving unsolvability may take longer. Indeed, plotting the runtimes for success and failure separately, demonstrates that (Figure 5). Interestingly, there are no failures due to timeout: Rewrite-SHOP3 is able to prove unsolvable all of the unrepairable problems). While the IPYHOPPER times are generally the best on average, its times vary more widely: see Figure 4.

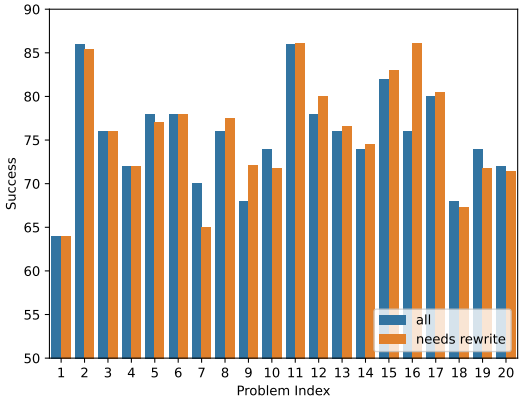

Figure 2: Rewrite-SHOP3 success rates for the satellite repair problems (note the y axis runs only from 50-90%).

Rovers The Rovers repair problems were more difficult than Satellite, and none of the repair methods solved all of them. Figure 6 shows success percentages. Note the outliers for IPYHOPPER and SHOPFIXER in problems 3, and 6 and for IPYHOPPER in problems 10 and 20. Gen-

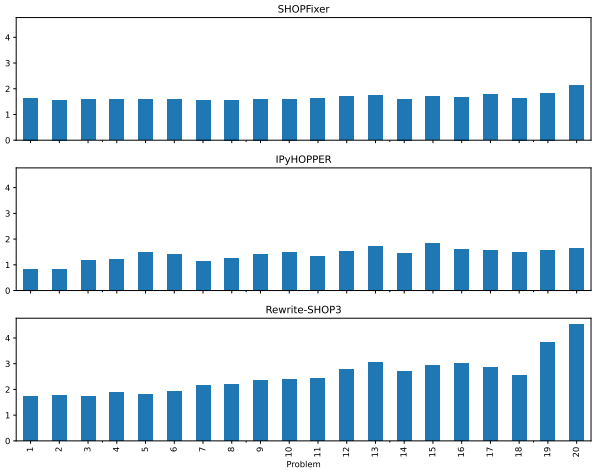

Figure 3: Satellite problem runtimes for all repair algorithms, in msecs (rounded up), plotted in $\log_{10}$.

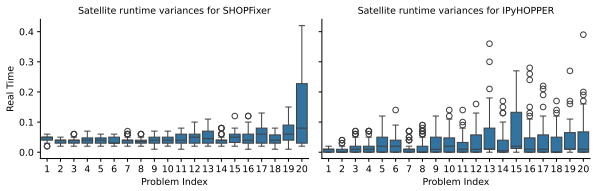

Figure 4: Variance of Satellite problem runtimes.

erally, SHOPFIXER is more successful than IPYHOPPER, as shown in Table 1.

Generally, while Rewrite-SHOP3 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 Rewrite-SHOP3 is only 84% (42/50) successful.

Runtimes are graphed in Figures 7, and 8. 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, IPYHOPPER is generally faster, but SHOPFIXER scales better with problem difficulty. For IPYHOPPER, problem 3 is an outlier in elapsed time, SHOPFIXER has issues with problem 6, and for Rewrite-SHOP3 the last two problems, and especially problem 20, are notably more difficult. As before, on the successful problems, IPYHOPPER has a higher runtime variance than SHOPFIXER (see Figure 9; variance plotted in seconds, not transformed).

Openstacks For Openstacks, both SHOPFIXER and Rewrite-SHOP3 solved all the repair problems. IPYHOPPER solved *almost* all, but failed for a small number (see Table 2). The runtimes, graphed in Figure 10 clearly show that Rewrite-SHOP3 and IPYHOPPER do not scale well on the more difficult problems, with Rewrite-SHOP3 notably worse. IPYHOPPER runtimes for problems 3, 6, and 12 are outliers: they are much more difficult than similarly-

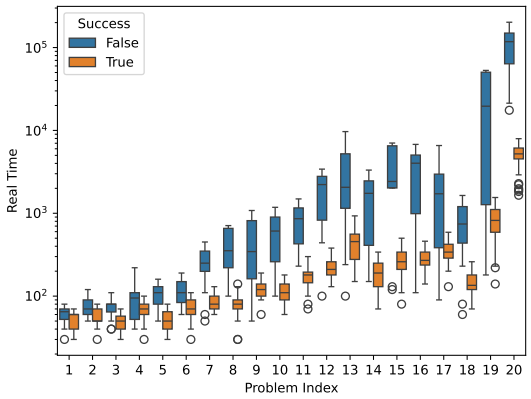

Figure 5: Satellite problem runtimes for Rewrite-SHOP3, comparing successful trials versus failed trials, plotted in $\log_{10}(\text{msec})$.

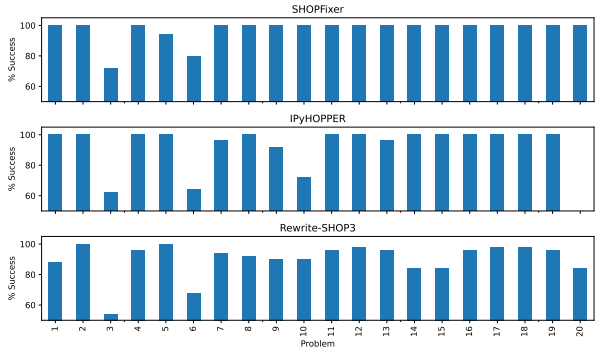

Figure 6: Success rates for the Rovers repair problems for each of the three algorithms.

numbered problems.

5 Discussion

Common Features Across all of the domains, REWRITE 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. Note that this could likely be substantially improved by heuristic guidance that would direct the early part of planning towards methods that replicate actions previously seen and that avoid infeasibly introducing new actions.

Satellite It is unsurprising that REWRITE 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 (see Section 2). Perhaps the surprise is that *any*

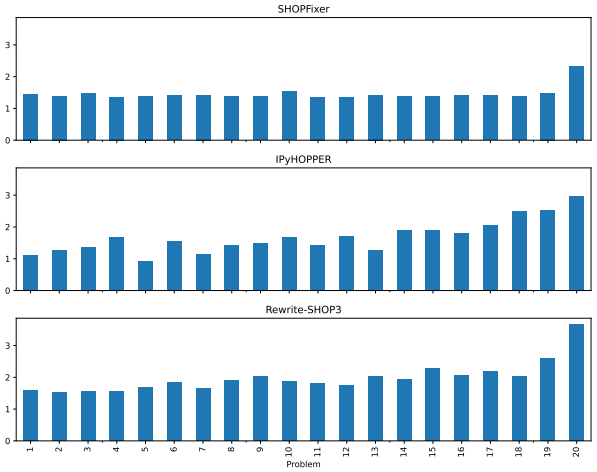

Figure 7: Runtimes for the Rovers repair problems in $\log_{10}(\text{msec})$ for each algorithm. These include only those problems solved successfully by the algorithm in question.

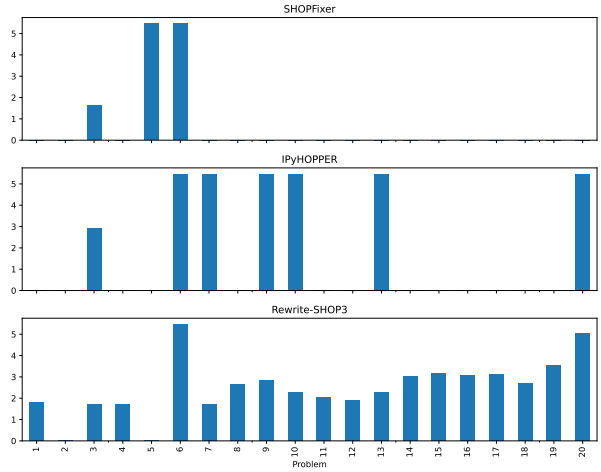

Figure 8: Runtimes for the Rovers repair problems in $\log_{10}(\text{msec})$ for each algorithm. These are only the problems *not* solved successfully by the algorithm in question.

	Solver	
Problem	IPyHOPPER % Success	SHOPFixer % Success
3	62%	**72%**
5	**100%**	94%
6	64%	**80%**
7	96%	**100%**
9	92%	**100%**
10	72%	**100%**
13	96%	**100%**
20	24%	**100%**

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

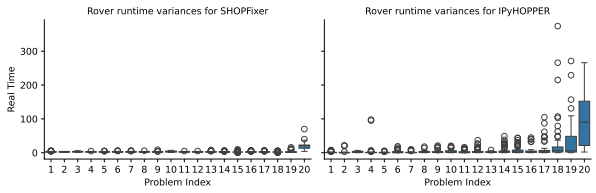

Figure 9: Variance in runtimes for the Rovers repair problems for IPyHOPPER and SHOPFixer. Only successfully solved problems are plotted.

of these scenarios are successfully repaired by Rewrite. We investigated further, and found that Rewrite 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 2.

Here is an example of how Rewrite'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, IPyHOPPER is generally faster than SHOPFixer, although it is implemented in interpreted Python rather than compiled Common Lisp. This is probably accounted for by the fact that SHOPFixer invests in building complex plan trees that include dependency information (see Figure 1) in order to more rapidly identify the location of precondition violations and their implications. SHOPFixer also uses this information to *backjump* (Dechter 2003; Gaschnig 1979) to the point of failure, instead of relying on chronological backtracking, as does IPyHOPPER. For these simple problems, SHOPFixer's added effort is generally not worthwhile. We note that the *variance* of IPyHOPPER's runtimes is wider than that of SHOPFixer, and that there are more outliers (Figure 4).

Rovers There were several Rovers problems where even IPyHOPPER and SHOPFixer 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, SHOPFixer'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. SHOPFixer's backjumping appeared to serve it well in the Rovers domain: IPyHOPPER had much more difficulty with these problems. It failed to repair 97 cases, of which 78 were due to timeouts. Time-

Problem	Success Rate
10	98%
19	98%
28	90%
29	92%
30	96%

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

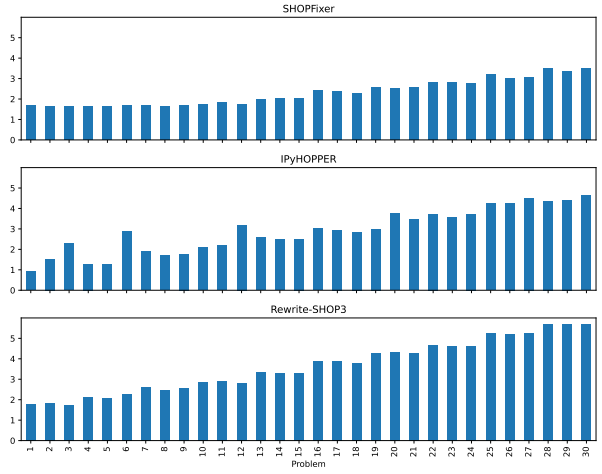

Figure 10: Runtimes for the Openstacks repair problems in $\log_{10}(\text{msec})$ for each algorithm. These include only those problems solved successfully by the algorithm in question.

outs 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 SHOPFixer's more expensive tree representation in runtime. We can see this even more clearly if we plot the two algorithms directly against each other (Figure 11). Indeed, IPyHOPPER'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 IPyHOPPER's simple chronological backtracking.

In a reversal of the previous patterns, Rewrite solves all of the problems. This is due to a difference in the way the domain was formalized compared to the other two domains. 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 Rewrite as did the fact that action choice is primarily constrained by preconditions, rather than by method structure, which also

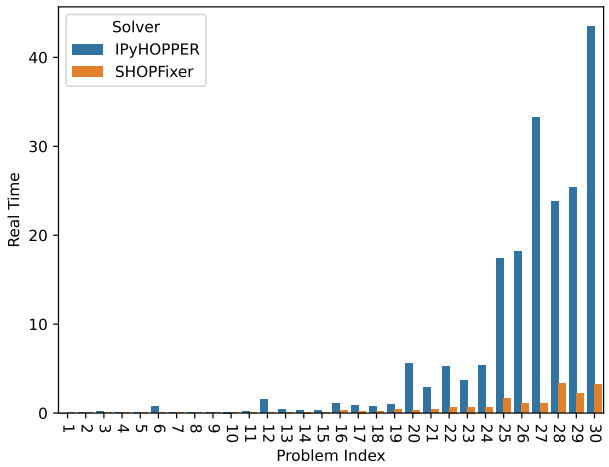

Figure 11: Direct comparison of runtimes for IPyHOPPER and SHOPFixer on Openstacks problems. Note that these are plotted in seconds, and not on a log scale.

avoided issues with this algorithm. Note that this relatively unconstrained planning also made it more difficult to generate the initial plans for this domain.

6 Conclusions and Future Directions

We have presented an analysis of three recent hierarchical repair algorithms from the AI planning literature; namely, SHOPFixer, IPyHOPPER, and Rewrite. Qualitatively, our analyses highlighted significant differences among these methods: First, Rewrite's definition of plan repair is more stringent than the others; we have identified benchmark planning problems that are solvable for IPyHOPPER and/or SHOPFixer that cannot be solved by Rewrite. Secondly, SHOPFixer attempts to detect when a plan will be invalid, before any actions actually fail; i.e., SHOPFixer invests in both data structures and computation in order to detect compromise to a plan as soon as possible. IPyHOPPER, on the other hand, 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 Rewrite 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 Rewrite in grounded planning systems. How best to schedule re-grounding *vis-à-vis* generation of the rewrit-

ten 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 studying 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), which are 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. Similar to *refineability* properties (Bacchus and Yang 1992; Yang 1997), this approach can be examined formally, theoretically, and experimentally.

Acknowledgments. 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.

References

Ayan, F.; Kuter, U.; Yaman, F.; and Goldman, R. P. 2007. HOTRiDE: Hierarchical Ordered Task Replanning in Dynamic Environments. In *ICAPS-07 Workshop on Planning and Plan Execution for Real-World Systems*.

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 the International Conference on Automated Planning and Scheduling*, volume 24, 386–394.

Cushing, W.; and Kambhampati, S. 2005. Replanning: A New Perspective. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 13–16. Monterey, CA USA: AAAI Press.

Dechter, R. 2003. *Constraint Processing*. San Francisco: Morgan Kaufmann Publishers. ISBN 978-1-55860-890-0.

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

Gaschnig, J. 1979. Performance Measurement and Analysis of Certain Search Algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University. Reference for backjumping, cited in Ginsberg's texbook.

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

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.

Hoffmann, J. 2011. Analyzing Search Topology without Running Any Search: On the Connection between Causal Graphs and h^+. *Journal of Artificial Intelligence Research*, 41: 155–229.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2021. The PANDA Framework for Hierarchical Planning. *KI - Künstliche Intelligenz*, 35.

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 the 34th AAAI Conference on Artificial Intelligence (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. ISBN 978-3-030-58284-5 978-3-030-58285-2.

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 International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research*, 20: 1–59.

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

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 6th ICAPS Workshop on Hierarchical Planning (HPlan 2023)*, 10–18.