Implicit Dependency Detection for HTN Plan Repair

Paul Zaidins[1,3], **Mark Roberts**[3], **Dana Nau**[1,2]

[1]Dept. of Computer Science and [2]Institute for Systems Research, University of Maryland, College Park, MD, USA
[3]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
[1]{pzaidins, nau}@umd.edu [3]{paul.zaidins,mark.roberts}@nrl.navy.mil

Abstract

Two recent approaches to HTN replanning, IPyHOP and SHOPFIXER, replan by adapting the previously planned solution when an action fails. IPyHOP replans the entire solution tree after the failure, while SHOPFIXER uses pre-calculated dependency graphs to replace portions of the tree; neither uses forward simulation of the plan to predict where future failures might occur.

This paper describes IPyHOPPER, which improves IPyHOP by retaining more of the information provided by the hierarchy and using forward simulation to repair minimal subtrees that contain future failures. Our experimental comparisons show that in domains where errors are not rare, IPyHOPPER is both faster and uses fewer iterations to repair than IPyHOP's repair mechanism. IPyHOPPER's repair speedups are similar to those of SHOPFIXER when given a probabilistic error model with nontrivial error rates.

1 Introduction

Given some level of domain expertise, a Hierarchical Task Network (HTN) can be leveraged to solve complex problems quickly. Hierarchy is perhaps the most powerful feature of HTN planners. For planning in static, known environments using this hierarchy is straightforward. However, because the world is often dynamic and full of uncertainty, acting must be properly coupled with *online* planning to operate well in such conditions. So the speed and efficiency of correcting errors in plans is important.

When a failure is encountered the most obvious solution is to replan using the current state as the new initial state. Even though it may be incorrect after a failure, the original plan provides useful information regarding the hierarchy and implicit action restrictions. Reusing portions of the original plan may also promote plan stability (Fox et al. 2006).

Two recent approaches to plan repair involve updating original solution tree: SHOPFIXER (Goldman, Kuter, and Freedman 2020) and the Lazy-Refineahead repair algorithm in the IPyHOP paper (Bansod et al. 2022). Both approaches use portions of the original plan when facing disruptions during execution, but differ in the following respects.

- Lazy-Refineahead uses the structure of the original plan up to the point of failure, but discards the original plan hierarchy past the failed action. In contrast, SHOPFIXER pre-calculates dependency graphs, so that when action ends in failure, the plan repair process can skip subtrees that are not dependent on the action that failed. This allows for minimal fixes, increasing plan stability and reducing repair time.
- SHOPFIXER uses actions and methods in the SHOP 3 format (Lisp code with similar structure to PDDL), so programmers must understand how to implement methods in that format. IPyHOP's methods and operators are Python functions, making them more accessible to those unfamiliar with automated planning formalisms. This makes the explicit dependency graph calculation, as SHOPFIXER does, impossible due to the lack of well-defined preconditions or effects.

This paper describes IPyHOPPER, which extends IPyHOP[1] to use plan repair techniques inspired by those in SHOPFIXER. IPyHOPPER improves IPyHOP's repair efficiency by retaining more of the information provided by the hierarchy and using forward simulation. Rather than cutting the solution tree at the failed node, it selectively prunes future solution branches using forward simulation to repair minimal subtrees containing future failures. This avoids future failures that may arise from the plan alterations from repairing the minimal subtree of the immediate failed action and alterations brought from prior preemptive repairs. Our contributions include:

1. A new algorithm, IPyHOPPER, that performs a mix of in-place repair with forward simulation;
2. Revised benchmarks to support evaluation of IPyHOPPER against IPyHOP and SHOPFIXER; and
3. Experiments showing that IPyHOPPER reduces planner iterations and execution time for the 5 tested domains while keeping similar, or sometimes better, plan costs.

The rest of the paper gives a brief overview of related work, presents our IPyHOPPER algorithm for minimal repair with IPyHOP, and describes a comparison of the performance of IPyHOPPER, Lazy-Refineahead, and SHOPFIXER.

[1]IPyHOP and IPyHOPPER can plan for both tasks and goals, but for brevity we will refer to both of them as HTN planners. Section 2.5 provides further justification for this terminology.

2 Related Work

Our work is based on the implementation of IPyHOP and we compare to SHOPFIXER, so we will discuss each of these before pointing out other related work.

2.1 IPyHOP

IPyHOP (Bansod et al. 2022), is an iterative, domain-independent, totally-ordered, Goal Task Network planner written in the Python programming language. Goals and tasks are ordered tuples while methods and actions are arbitrary Python code blocks. Input goals and tasks are repeatedly decomposed until only actions remain. This results in a solution tree from which the plan can be read as the preorder traversal of actions (i.e., leaves) of the tree.

IPyHOP's repair algorithm, Lazy-Refineahead, works by finding the failed action in the solution tree, removing all nodes to the right of the failed action in the preorder traversal, and then marking nodes that lost children. From there the plan is repaired by iterating through the tree and redoing decomposition at those marked nodes with the current state used for the preorder traversal's leftmost new nodes. It should also be noted that method preconditions are only checked during planning and during plan repair for methods touched during repair.

2.2 SHOPFIXER

SHOPFIXER (Goldman, Kuter, and Freedman 2020) is a plan repair system for the SHOP 3 (Goldman and Kuter 2019) HTN planner. Unlike IPyHOP, SHOP 3 uses highly structured Lisp code for methods and actions. Variables are instantiated primarily through unification. This allows for the computation of causal links in the solution tree. SHOPFIXER uses these causal links to isolate plan repair to only the sections of the solution tree that are relevant. This served as part of the inspiration for IPyHOPPER, the principal difference being that our methodology does not impose any structural restrictions on the decomposition methods and actions, as dependencies are found implicitly through simulation rather than explicitly computed. Thus, IPyHOPPER can still speed up plan repair even when a dependency graph is not known.

2.3 Plan Repair by Domain Modification

Given a planning domain D, task t and initial state s_0, suppose an HTN planner returns a solution plan $\pi = (a_1, \ldots, a_i, \ldots, a_n)$. While executing π, suppose an execution error occurs at a_i, producing a state s'_i rather than the predicted state s_i. Höller et al. (2020) define a modified planning domain D' in which (i) the predicted outcome of a_i is s'_i and (ii) given D', t, and s_0, the HTN planner returns a solution plan π' such that the first i actions are the same as in π. Thus if no further errors occur, t can be accomplished by executing the part of π' that starts at the action after a_i.

Unlike IPyHOPPER, π' preserves none of the unexecuted part of π. We think the actions in π' after a_i are the same as Lazy-Refineahead would have produced. Technically this can be viewed as a hybrid approach between plan repair and replanning.

2.4 Other Approaches

There has been work speeding up replanning by reusing plan solution trees (Soemers and Winands 2016). This differs from standard plan repair in that existing plans are modified for new tasks as opposed to altering an existing plan for the same task given an interruption. RepairSHOP (Warfield et al. 2007) uses a directed, dependency graph know as a goal graph to track alternative decisions (decompositions and instantiations). When plan repair is needed, the planner is reset to the planning state of the first applicable alternative found. HOTRiDE (Ayan et al. 2007) takes a similar approach with its task-dependency graph.

2.5 Evolution of HTN Terminology

Some of the best-known early formulations of HTN planning included both goals and tasks (Currie and Tate 1991; Kambhampati and Hendler 1992; Erol, Hendler, and Nau 1994). However, SHOP and its successors used a simplified HTN formulation that omitted goals (Nau et al. 1999, 2001; Goldman and Kuter 2019). Their popularity led researchers to lose track of goals as a part of HTN planning, and HGNs were subsequently conceived as separate from HTNs (Shivashankar et al. 2012, 2013). In this paper, we return to the earlier concept that HTN planning includes both goals and tasks.

3 Repairing Minimally with IPyHOPPER

Before we formally describe IPyHOPPER, we will contrast it with an example from the IPyHOP paper. Figure 1(a) copies Figure 1 by Bansod et al. (2022). It represents a notional hierarchical plan for tasks $t1$, $t2$, and $t3$. Hexagonal nodes indicate method instances m_i for a task t_j, and rectangular nodes indicate an operator instance o_i. The resulting plan, $\pi = \langle o1, o2, ..., o11, o12\rangle$ is produced using a Depth First Search (DFS) tree preorder traversal. Moreover, $o7$ produces effects on which $o11$ relies, shown as a red dashed line; this will be an important detail in the following comparison.

While executing π, $o7$ nondeterministically fails. The Lazy-Refineahead algorithm discards the plan structure for the parent of the failed node as well as the nodes to the right of the failed node in preorder traversal, which includes $m1\_t4$, $m1\_t5$, and $m1\_t3$. This results in nine nodes removed from the tree. But it might be the case that only a few of these need to be changed to repair the plan.

Instead, IPyHOPPER preserves as much of the tree as possible to minimize computation and maximize stability (Fox et al. 2006). It does this by combining forward simulation (i.e., action application to each state) with localized repair. As simulation progresses forward, each action is checked for applicability. If the simulation succeeds for all actions in the remaining plan, then the repair was localized to the failed node. When the simulation fails for an *existing* action in a plan, that is treated as a potential future failure, resulting in further repair.

To make this concrete, consider Figure 1(b), where a dashed green line indicates starting point of simulation with, $t4$, the parent of the failed action having been unexpanded. IPyHOPPER removes only the children of $t4$ to find a new decomposition the parent of the failed action using the current state, resulting in two new actions.

IPyHOPPER simulates forward from $o13$ to check for any future problems from the repair. Forward simulation reveals that actions $o8$, $o9$, and $o10$ will succeed so the tree supporting these actions remain in the plan. The result is shown in Figure 1(c) between the green and orange dashed lines. At this point, suppose $o11$ fails, perhaps because its precondition from $o7$ was not met by the new actions $o13$ and $o14$. Figure 1(d) shows that IPyHOPPER makes another repair to rightmost instance of $t4$, resulting in a new action $o15$. From the original plan, actions $o8$, $o9$, and $o10$ remained. We include Figure 2 to demonstrate the difference in the repair process with Lazy-Refineahead. Note how the parent and all pre-order succeeding nodes are unexpanded regardless whether the actions could still have been performed.

To summarize, IPyHOPPER repairs a plan by removing a failed node's immediate subtree then repairing while simulating possible future failures. Using the new observed state, it fixes the tree at the point of failure, traversing upward as needed while simulating forward. If the simulation yields errors, it fixes the next point of failure and continues simulating forward. This repeats until it reaches an unrecoverable state or the end of the simulated plan without a failure. A repair-simulate cycle might introduce further failures. This occurs when failure points have a common ancestor and the rightmost failure point has no applicable repair given the previous failure point repair. In such a case, it backtracks to the previous failure point. Backtracking to the root node of our tree indicates no plan is possible. Otherwise, it returns the repaired plan.

3.1 The Repair Algorithm

Algorithm 1 formalizes the above example. We assume here that the root of the full tree exists as the parent of all tasks provided to the planner as in IPyHOP.

Lines 2 and 3 place the parent of the failed action and current state into the stack and enters the repair-simulate loop. This loop continues until either simulation results in a successful plan or the algorithm reaches an unrecoverable state.

Lines 5 and 6 read the top node and state from the stack and replace the top node with its parent. This results in either moving up the solution tree when repair fails for the current subtree or moving to the previous repair when arriving at the root.

Line 7 removes the current decomposition of f to allow for a new decomposition. This begins the minimal repair, from which Lines 8 to 12 will check to see if any decomposition methods are relevant and if so will attempt to repair the subtree of w rooted at p.

The expansion is performed using a modified IPYHOP planning mechanism. The most important changes are preventing planning to progress to the top node of a subtree (thus allowing targeted subtree repair) and altering IPyHOP methods to find all potential variable bindings rather than a single binding (thus allowing functionality similar to unification). There are two cases to consider: (1) If there are relevant methods, but no applicable expansions, return to the top loop at Line 4. (2) If no relevant actions exist, check if p is a common ancestor of any other node in the stack and, if so, remove p and its associated state from the stack as the next loop will clobber previous repairs. Either case results in a return to the top of the loop.

A successful repair of the subtree of w results in advancing the simulation at Lines 17 to 25. Simulation proceeds by reading the suffix of π starting from w and following it to completion from our current state. At this point, there are two cases to consider. (1) If π has another failure, s' becomes the state at failure, the parent of the action that failed is placed on the stack, and the loop returns to the top. (2) If the simulation completes the rest of π without failure we terminate the loop and return π as the repaired plan to begin executing. We note here that the stack thus functions as our backtracking mechanism as the stack consists of the parents of our repaired nodes. We exhaust all possible repairs for our current node before returning to the immediate prior repair point to look for a solution If the stack should become empty (i.e., the algorithm arrive at the root), then tree must have been stripped down to the root and π will be empty. When this occurs, an empty plan indicates failure of the algorithm to produce a viable repair.

Algorithm 1: IPyHOPPER plan repair algorithm.

```
1  Def IPyHOPPER( state: s, decomposition tree: w, failed action node: f ):
2      $p \leftarrow$ parent of $f$ in $w$;
3      $stack \leftarrow [(p, s)]$;
4      while stack not empty do
5          $f, s \leftarrow$ pop($stack$);
6          $p \leftarrow$ parent of $f$ in $w$;
7          unexpand $f$;
8          if $f$ has possible decomposition then
9              $subTree \leftarrow$ subtree of $w$ with root $f$;
10             attempt expansion of $subTree$ from $s$;
11             if $subTree$ cannot be fully expanded then
12                 continue;
13         else
14             if if the parent p is an ancestor of a previous node then
15                 pop($stack$);
16             continue;
17         $\pi \leftarrow$ plan from $w$;
18         simulate($s$, $\pi$);
19         if simulation failed then
20             $s' \leftarrow$ input state for failed action;
21             $p_a \leftarrow$ parent of failed action node;
22             push(stack, $(p_a, s')$);
23             continue;
24         else
25             break;
26     $\pi \leftarrow$ plan from $w$;
27     return $\pi$;
```

4 Experiments

We incorporated Algorithm 1 into Run-Lazy-Refineahead with no other changes, resulting in IPyHOPPER. We compared IPyHOPPER to IPyHOP as well as to SHOPFIXER.

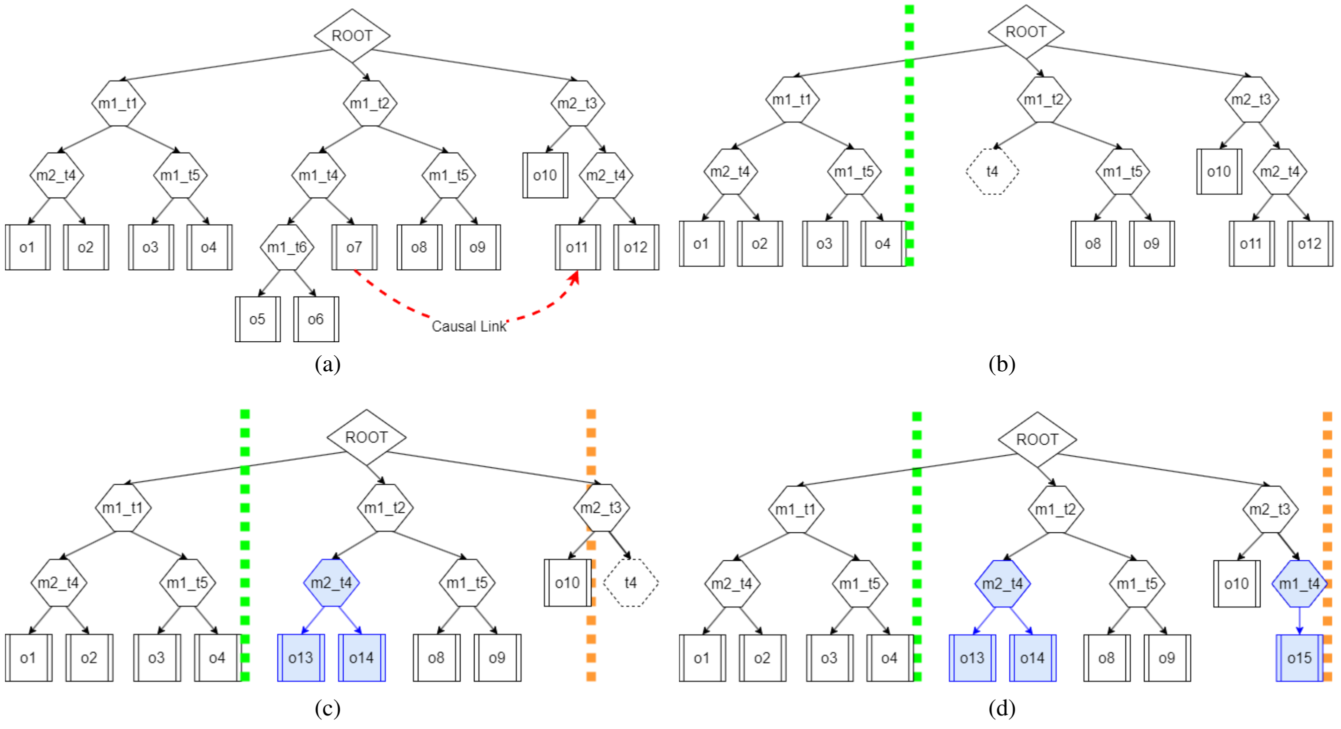

Figure 1: An example IPyHOPPER repair using Figure 1 by Bansod et al. (2022). See prose for a detailed description.

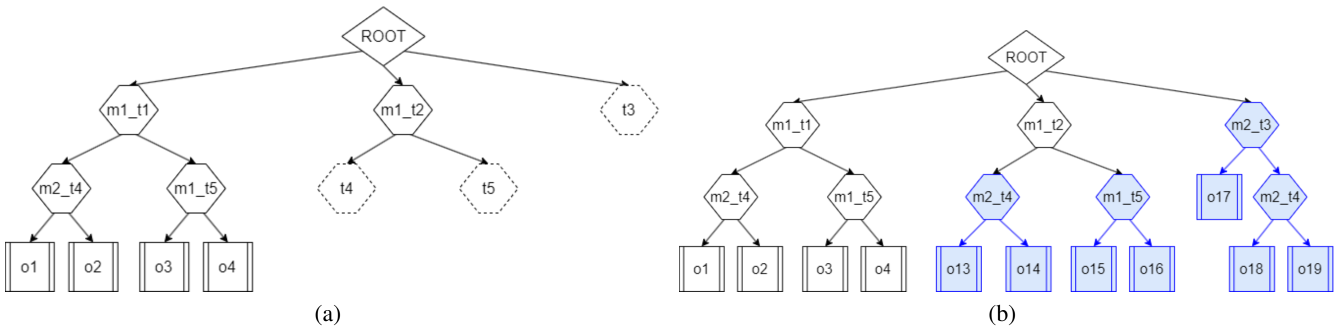

Figure 2: With Lazy-Refineahead, 1(b), 1(c), and 1(d) would have been the above.

4.1 IPyHOP Comparison

To evaluate how well IPyHOPPER improves the performance of IPyHOP, we repeat the study on two domains from the IPyHOP paper (Bansod et al. 2022).

The robobsub domain comes from the RoboSub 2019 competition. An autonomous submersible is expected to complete a course consisting of several tasks. Not all tasks must be completed for an attempt at the course to be considered complete, but these optional tasks yield additional points. For such tasks non-empty decompositions are attempted first by the internal ordering of the methods and if no non-empty decomposition is applicable an empty decomposition is returned rather than concluding no plan is possible.

The rescue domain consists of several unmanned air and ground vehicles attempting to perform search and rescue operations following some disaster. One important difference between these domains is that the rescue domain has states from which recovery is impossible.

The experimental setup is identical except for more runs; the starting seeds increased from 1,000 to 10,000 with number of trials per seed increased from 11 to 100. We evaluate using two metrics from Bansod et al. (2022). *Total decompositions* assesses the total number of nodes expanded, thus tasks and actions are both counted, while *total action cost* is the sum of the costs of all actions attempted, including failed and successful actions.

For the Rescue domain, we found that of our 10,000 seeds, 412 were unsolvable. Of the remaining 9,588 configurations, 82 of them had at least one trial end in an unsolvable state when using the Lazy-Refineahead algorithm. IPyHOPPER had no such partial seed failures.

Rescue IPyHOPPER produces plans with lower total action cost using fewer node expansions. Figure 3(a) shows that IPyHOPPER produces a lower total action cost for most problems, resulting in an averaged reduction of 2.28 ± 0.02 with 95% confidence. This is most likely a domain-specific effect, as the algorithm does not impose any cost-related conditions. Most plan costs occur in discrete, relatively narrow bands while IPyHOP produces a much less coherent pattern. This is likely a consequence of the stability that the new algorithm imposes through minimal plan changes.

Figure 3(b) shows that IPyHOPPER expands significantly fewer nodes than Lazy-Refineahead for all configurations (28.77 ± 0.03 with 95% confidence). This is in line with expectations, as the failures in this domain are generally fixable by repairing only the immediate parent of the failed node, while Lazy-Refineahead must replan all of the unexecuted plan.

Robosub IPyHOPPER produces plans with higher total action cost using fewer node expansions. Figure 3(c) shows that IPyHOPPER produces plans with a significantly higher total action cost for all problems (33.80 ± 0.02 with 95% confidence). This is unexpected, but perhaps a consequence of the unique feature of most actions being skippable in this domain. Optional objectives are rewarded with additional points, but are not required. The method of last resort to decompose such tasks is often simply to return nothing.

Table 1: Mean percentile change in paired sample difference for CPU time and mean iteration count by domain. A negative value indicates that IPyHOPPER is outperforming Lazy-Refineahead.

Domain	Change in mean CPU time (%)	Change in mean iteration count (%)
Openstacks	-34.5	-16.1
Rovers	-40.7	-49.3
Satellites	-34.0	-49.3

Figure 3(d) shows that IPyHOPPER not only expands significantly fewer nodes ($59.45 \pm 0.07\%$ with 95% confidence), but also reveals a nearly constant number of node expansions. This suggests IPyHOPPER is exceptionally stable for this domain.

4.2 SHOPFIXER Comparison

To compare IPyHOPPER to SHOPFIXER, we compare the difference between Lazy-Refineahead and IPyHOPPER on the SHOPFIXER domains explored by Goldman and Kuter (2020): openstacks, rovers, and satellites.

Our first task was to adapt the SHOP 3 methods as faithfully as possible. This is somewhat challenging because the exact format of the state is more ”Pythonic” and IPyHOP does not use unification. We also needed to enable IPyHOP methods to return multiple different instantiations based on ground arguments. Otherwise, we replicated the spirit of the SHOP 3 methods, actions, and deviations.

We ran each problem 1000 times with a nominal error rate of 10%. To replicate the potential error distribution from the SHOPFIXER experiments while allowing for multiple errors in a single experiment, we scaled each action's likelihood of error linearly in proportion to the number of potential errors for that action and the current state.

Errors were introduced randomly and are uniformly randomly selected from all potential errors for that action and the current state. In general this meant that the action-state pair with the largest number of potential errors (as calculated by a running max) would have some error occur with probability equal to the nominal probability. Time spent calculating execution errors was not included in time measures for the planner. All domains here had only recoverable failures, so we did not need to consider the case of failed planning.

We recorded three metrics: action count, CPU time, and iteration count. *Action count* is the number of all actions attempted, thus both failed and successful actions are counted. *CPU time* is the elapsed process time from immediately prior to the initial call to the planner for a plan to immediately after the successful completion of the last action. *Iteration count* is the total number of iterations the planner runs for both in initial planning and every call to the plan repair algorithm. Iteration count includes iterations spent traversing through the solution tree without expanding nodes, and is slightly different than node expansions.

Openstacks IPyHOPPER has similar performance in terms of action counts but generally uses less CPU time and fewer

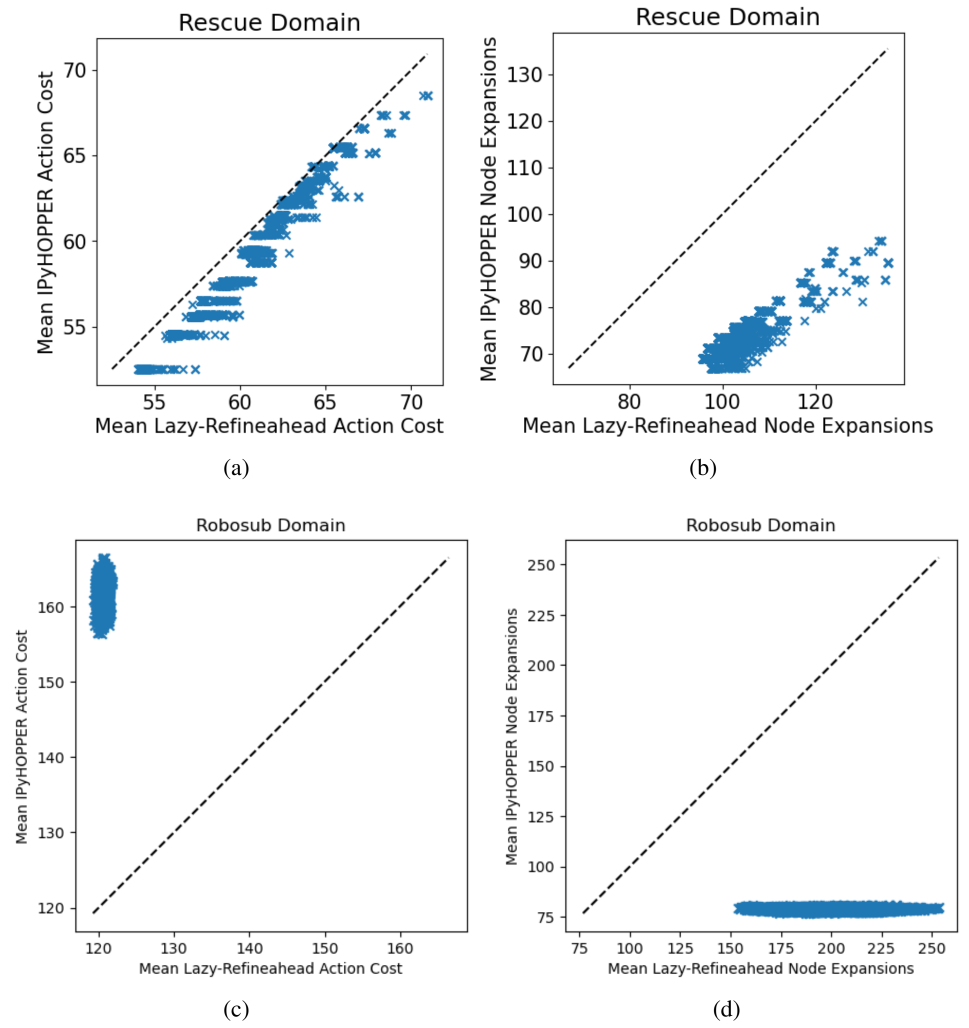

Figure 3: Scatter plots showing the mean action cost and node expansions for the Lazy-Refineahead and IPyHOPPER algorithms, in the Rescue and Robosub domains. For the metrics shown, points below the dashed line represent IPyHOPPER performing better than Lazy-Refineahead.

iterations. Figure 4 shows the spread for IPyHOPPER is no larger than SHOPFIXER and, for the larger problems, is significantly smaller in iteration count and CPU time. This comes from the increased stability of the new algorithm. There is a notable decrease in the medians of the iteration count and CPU time distributions when looking across all problems. However, this varies across problems. Action counts are essentially identical between the algorithms.

Rovers IPyHOPPER produces more varied plan costs but does so with lower CPU time and fewer iterations. Figure 5 shows a large improvement in iteration count and CPU time for most problems, especially for the largest problems. There is still a reduction in spread for iteration count and CPU time. There is definitely some distribution shift in the action count, but appears to highly dependent on problem.

Satellites IPyHOPPER produces results similar to Rovers with sometimes large improvements in action count using reduced iteration count or CPU time. Figure 6 shows less of a shift in the action count distributions. Both the rovers and satellites domain involve multiple heterogeneous agents with occasionally redundant capabilities fulfilling goals with essentially no ordering constraint. This sort of problem is amenable to the IPyHOPPER because repairs of a single subtask are localized in the solution tree.

Summary In Table 1 we summarize the performance gains of IPyHOPPER over Lazy-Refineahead in the SHOPFIXER domains. We find that for all three domains there is a significant improvement in both CPU time and iteration count.

5 Conclusions and Future Work

IPyHOPPER offers substantial improvement in iteration count and computation time across all 5 tested domains. Given the diverse nature of these domains, we speculate that

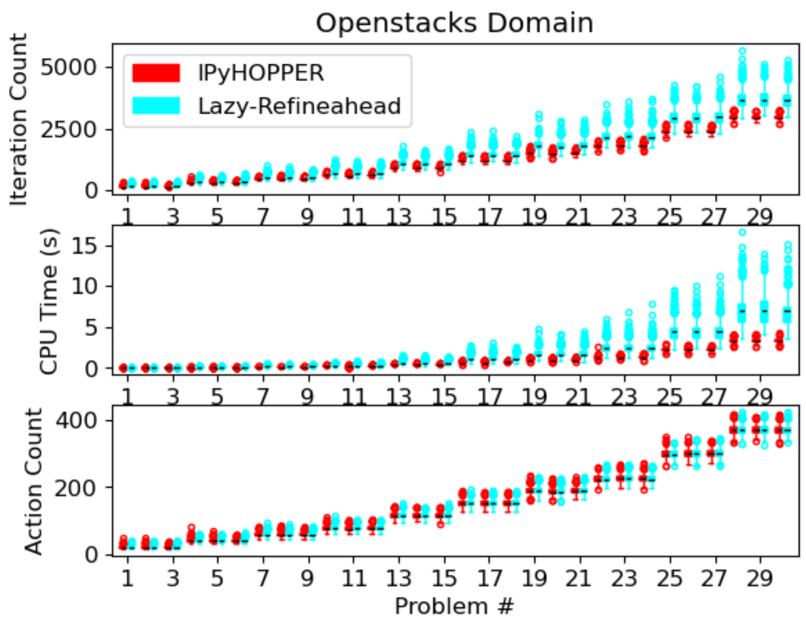

Figure 4: Key metric distributions for openstacks domain.

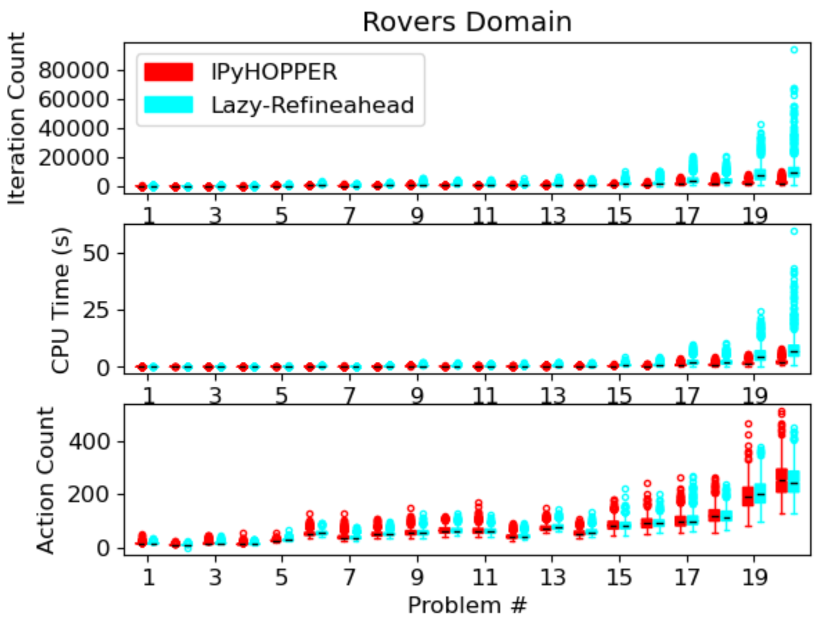

Figure 5: Key metric distributions for rovers domain.

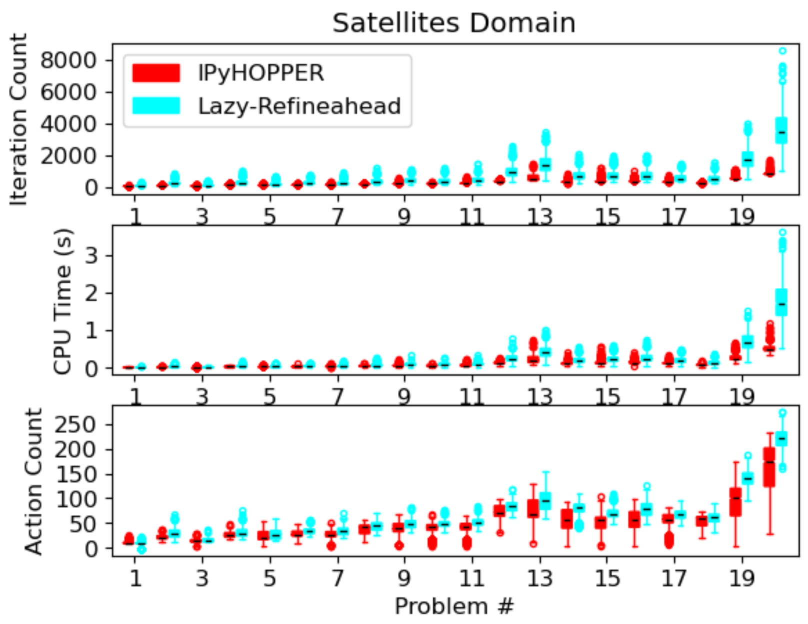

Figure 6: Key metric distributions for satellites domain.

this approach could yield similar benefits for many domains of interest. However, this approach cannot work as-is for all planning domains, because certain conditions regarding the definition of actions, methods, and the nature of errors must be met for plans to be guaranteed to be correct. These conditions are not currently well defined and this could be a subject of future work.

There seems to be a general increase in the relative benefit of the new algorithm with problem size. However, this needs to be investigated further because our suspicion is that the tested domains may be relatively sparse in terms of how subtasks are related. A better understanding of this quality of separability may yield valuable insights into planning. As the requirement for explicit preconditions and effects is removed in IPyHOPPER, it may offer some transferable lessons for refinement and hybrid plan repair.

Our experiments were in domains expressed in the well known PDDL. This does not make use of the full expressitivity possible with IPyHOPPER. Given IPyHOP should be capable of planning in domains not well expressed in PDDL form, we would like to measure potential gains in such domains.

Acknowledgements

This work has been supported for UMD in part by ONR grant N000142012257 and AFRL contract FA8750-23-C-0515. MR thanks ONR and NRL for funding this research. The views expressed are those of the authors and do not reflect the official policy or position of the funders.

References

Ayan, N.; Kuter, U.; Yaman, F.; and Goldman, R. 2007. HOTRiDE: Hierarchical ordered task replanning in dynamic environments.

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

Currie, K.; and Tate, A. 1991. O-Plan: The Open Planning Architecture. *Artificial Intelligence*, 52(1): 49–86.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In *AIPS*, 249–254.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: replanning versus plan repair. In *Proceedings of the Sixteenth International Conference on International Conference on Automated Planning and Scheduling*, 212–221. AAAI Press. ISBN 978-1-57735-270-9.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: the case of SHOP3. In Neuss, N., ed., *Proceedings of the 12th European Lisp Symposium (ELS 2019), Genova, Italy, April 1-2, 2019*, 73–80. ELSAA. ISBN 978-2-9557474-3-8.

Goldman, R. P.; Kuter, U.; and Freedman, R. G. 2020. Stable Plan Repair for State-Space HTN Planning. In *ICAPS Workshop on Hierarchical Planning (HPlan)*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020. HTN Plan Repair via Model Transformation. In *KI 2020: Advances in Artificial Intelligence: 43rd German Conference on AI, Bamberg, Germany, September 21–25, 2020, Proceedings*, 88–101. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-030-58284-5.

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

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. In *IJCAI*, 968–973.

Nau, D. S.; Muñoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-Order Planning with Partially Ordered Subtasks. In *IJCAI*, 425–430. Seattle.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. 2013. The GoDeL Planning System: A More Perfect Union of Domain-Independent and Hierarchical Planning. In *IJCAI*, 2380–2386.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A Hierarchical Goal-Based Formalism and Algorithm for Single-Agent Planning. In *AAMAS*, 981–988.

Soemers, D. J. N. J.; and Winands, M. H. M. 2016. Hierarchical Task Network Plan Reuse for video games. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8.

Warfield, I.; Hogg, C.; Lee-Urban, S.; and Muñoz-Avila, H. 2007. Adaptation of Hierarchical Task Network Plans. In Wilson, D.; and Sutcliffe, G., eds., *Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference, May 7-9, 2007, Key West, Florida, USA*, 429–434. AAAI Press.