Stable Plan Repair for State-Space HTN Planning

Robert P. Goldman and **Ugur Kuter** and **Richard G. Freedman**
SIFT, LLC
319 1st Ave N., Suite 400,
Minneapolis, MN 55401, USA
{rpgoldman, ukuter, rfreedman}@sift.net

Abstract

This paper describes our approach, SHOPFIXER, to plan repair in Hierarchical Task Network (HTN) planning. We developed SHOPFIXER in the SHOP3 HTN planning framework, extending SHOP3's HTN language and theorem-proving capabilities in several ways. Unlike many existing HTN plan repair approaches that depend on chronological backtracking, SHOPFIXER uses backjumping techniques to efficiently, correctly and stably repair the hierarchical plans, where stability means with minimal perturbation to the original plan. We describe our new plan repair method and present experimental results in a number of IPC domains, demonstrating that it generates plans with limited perturbations, and that its plan repair is more computationally efficient than replanning. We compare our results with earlier experimental results from Fox, *et al.* on plan repair and plan stability. Our results confirm theirs, and generalize them. Specifically, we generalize their LPG-repair algorithm to handle plan upsets during execution, and evaluate it in such situations.

1 Introduction

Plan repair has been shown to provide advantages over generating new plans from scratch both in terms of planning runtime and in terms of plan *stability* – the amount of plan content that is retained between the original and repaired plans (Fox et al. 2006). Fox et al. demonstrated that plan repair could provide new plans faster, and with fewer revisions, than replanning *ab initio* in the face of disruptions. They use the term "stability" to refer to the new plan's similarity to the old one, by analogy to the term from control theory. In other prior work, the term "minimal perturbation" has been used synonymously. Plan stability is particularly important for human interaction: users are confused by radical changes to plans introduced in response to trivial upsets.

Previous work on plan stability was limited in that it could only handle plan upsets introduced by modifications to the initial state of the plan (Fox et al. 2006). Our work extends Fox et al.'s approach based on methods from existing plan-repair works such as Wilkins and desJardins (2001), Ayan et al. (2007), Bidot, Schattenberg, and Biundo, and Kuter (2012) in order to handle disturbance that can occur anywhere in the course of plan execution. One way in which SHOPFIXER is less general than Fox et al.'s is that it does not handle goal modification. We chose not to do this because we did not have a good metric for the size of a goal modification, and it is even less well supported by PDDL than are disturbances. Also, most other repair work is limited to disturbance handling.

In this paper, we describe SHOPFIXER, a new method for repairing plans generated by the forward-searching HTN planner, SHOP3. Our method uses a graph of causal links and task decompositions to identify a minimal subset of the plan that must be fixed in plan repair. We also extend the notion of plan repair *stability* introduced by Fox et al. (2006), and further develop their methods and experiments, which demonstrated the advantages of plan repair over replanning.

Contributions of this work are as follows:

- We describe a new technique, SHOPFIXER, and our extensions to the well-known totally-ordered HTN planning formalisms (Ghallab, Nau, and Traverso 2004; Goldman and Kuter 2019). This approach uses causal links for plan repair in SHOP3. Unlike previous such methods (Ayan et al. 2007; Kuter 2012) of the same vein, we show that plan repair and stability in SHOPFIXER is sound and complete, and that its flaw detection method is complete, but unsound: a conservative over-approximation.
- We show how to use NCD (Normalized Compression Distance) for evaluating plan stability and demonstrate how this measure refines plan stability significantly over the existing use of *action distance* (Srivastava et al. 2007) in (Fox et al. 2006) (see Figure 5).
- To support experimentation, we have extended Fox et al.'s LPG variant ("LPG-repair") to be able to repair plans upset in the middle of execution: previously it was only able to react to changes in the initial state, conceptually between the completion of planning, and the commencement of execution.
- In three different planning benchmark domains with different characteristics and over 700 planning and plan-repair planning problems in total, our experimental results confirm earlier results on the *general* advantages of plan repair over replanning, both in terms of computational effort, and in terms of plan stability. SHOPFIXER shows more stable and consistent effect on plan stability measured by NCD than does LPG-repair since SHOPFIXER is a lifted planner and can find plan modifications that are not distinguishable to LPG-repair (cf. Figures 6 and 7).

Listing 1: Effects expressions

```
<literal>* |
(forall (<variable>?) <literal>*) |
(when <GD> <literal>*),
```

- Our experimental results support the utility of our replanning method for forward HTN planning. The SHOPFIXER method is somewhat at odds with the method of Höller et al. (2018), which uses a subtly different definition of HTN plan repair. We discuss these differences with the pros and cons of both approaches.

2 Preliminaries

Our work is done in the context of the SHOP3 HTN planner (Goldman and Kuter 2019), successor to the earlier HTN planners, SHOP2 (Nau et al. 2003), and SHOP (Nau et al. 1999). Ghallab, Nau, and Traverso (2004) describe a restricted case of HTN planning called *Total-order Simple Task Network* (TSTN) planning, a formalization of the original SHOP HTN planning algorithm.[1] TSTN is a restricted version of HTN planning in which each method's subtasks are totally ordered, each method's constraints consist solely of preconditions, and no critics are allowed.

TSTN planning is defined over a language, L, the set of all literals in a function-free first-order language over a finite domain of quantification, $\omega(L)$. A *state*, $s \in S(L)$ is an assignment of truth values to every positive literal in L.

A *TSTN Planning domain* for a language, L, is a tuple[2]: $\mathcal{D}(L) = \langle \text{tasks}(\mathcal{D}(L)), \text{ops}(\mathcal{D}(L)), \text{meths}(\mathcal{D}(L)) \rangle$ of a set of *tasks*, *operators*, and *methods*. Each operator $o \in \text{ops}(domain)$ is a triple

$$o = (\text{name}(o), \text{precond}(o), \text{effects}(o)),$$

where name(o) is a *task* (see below), and precond(o) and effects(o) are sets of literals called o's *preconditions* and *effects*. An *action* α is an instance of an operator. If a state s satisfies precond(α), then α is *executable* in s, producing the state $\gamma(s, \alpha) = (s - \{\text{all negated atoms in effects}(\alpha)\}) \cup \{\text{all non-negated atoms in effects}(\alpha)\}$.

In SHOPFIXER, we extend this definition to PDDL2.1 actions, but without functional and arithmetic expressions. precond(o) is a logical expression as defined as `<GD>` in PDDL2.1 grammar (Fox and Long 2003). effects(o) is defined as in PDDL 2.1 excluding durative and functional expressions: see Listing 1.

A *task* is a symbolic representation of an activity. Syntactically, it is an expression $\tau = t(x_1, \ldots, x_q)$ where t is a symbol called τ's *name*, and each x_i is an element of $\omega(L)$. If t is also the name of an operator, then τ is *primitive*; otherwise τ is *nonprimitive* (or "complex"): $\text{tasks}(\mathcal{D}) = \text{prims}(\mathcal{D}) \cup \text{comps}(\mathcal{D})$. Intuitively, primitive tasks can be instantiated into actions, and nonprimitive tasks need to be decomposed (see below) into subtasks.

[1]SHOP3 and SHOP2 are more expressive extensions of SHOP.

[2]We leave the language arguments implicit in the future.

A *method*, m, specifies how to decompose a task:

$$m = \langle \text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m) \rangle$$

where name(m) is m's name and argument list, task$(m) \in \text{tasks}(\mathcal{D})$ is the task m can decompose, precond(m) is a set of preconditions, and subtasks$(m) = (t_1, \ldots, t_j), t_i \in \text{tasks}(\mathcal{D})$, the *expansion* of task(m), a sequence of subtasks.

Typically, we work with domain *descriptions*, collections of task, operator, and method *schemas*, with variables for some of the name, precondition, and effect arguments.

A *TSTN planning problem* is a three-tuple $P = \langle s_0, T_0, \mathcal{D} \rangle$, where s_0 is an initial state, T_0 is a sequence of ground tasks, the *initial task list*, and $\mathcal{D}$ is a TSTN domain.

If T_0 is empty, then P's only solution is the empty plan $\pi = \epsilon$, and π's *derivation* (the sequence of actions and method instances used to produce π) is $\delta = \epsilon$. We say that δ is executable in the current state, yielding no state changes.

If T_0 is nonempty (i.e., $T_0 = \langle t_1, \ldots, t_k \rangle$ for some $k > 0$), and s_0 is the current state, then let $T' = \langle t_2, \ldots, t_k \rangle$. If t_1 is primitive and there is an action α with name$(\alpha) = t_1$, and if α is executable in s_0 producing a state s_1, and if $P' = \langle s_1, T', \mathcal{D} \rangle$ has a solution π with derivation δ, then the plan $\alpha \bullet \pi$ is a solution to P (where $\bullet$ is concatenation) whose derivation is $\alpha \bullet \delta$. In that case, $\alpha \bullet \delta$ is executable in s_0. If t_1 is nonprimitive and there is a method instance m such that task$(m) = t_1$, and if s_0 satisfies precond(m), and if $P' = \langle s_0, \text{subtasks}(m) \bullet T', \mathcal{D} \rangle$ has a solution π with derivation δ, then π is a solution to P and its derivation is $m \bullet \delta$. The nonprimitive task task(m) is executable, if s_0 satisfies precond(m) and its subtasks(m) are executable in s_0. By induction, the derivation $m \bullet \delta$ is executable in s_0, if δ is executable. A sequence of tasks $t_1 t_2 \ldots t_n$ is executable in s_0 if t_1 is executable in s_0, the state after executing t_1 is s_1, and $t_2 \ldots t_n$ is executable in s_1.

The definition of a derivation, above, defines a *derivation tree* (or, less formally, a *plan tree*), with edges from complex tasks to the subtasks in their expansion, and with the primitive tasks of the plan as their leaves. We extend the plan trees by adding cross edges from primitive tasks that establish literals to the nodes that consume them in preconditions. While the establishers are all primitives, the consumers may be either primitives or complex tasks (note that the complex task as such does not consume the precondition, it is the *method* whose task network is used that has the preconditions).

A *plan disturbance* is an unexpected change in the world state after the execution of a prefix of the plan. A plan disturbance $\xi(\pi)$, for a TSTN plan, $\pi = \langle \alpha_1 \ldots \alpha_n, \rangle$ is a tuple $\xi(\pi) = \langle \text{pred}(\xi(\pi)), \text{effects}(\xi(\pi)) \rangle$, where $\text{pred}(\xi(\pi)) = \alpha_k$ is an action in π[3] and effects$(\xi(\pi))$ is a set of ground effects as in a STRIPS operator.

We assume an execution model that is a very simple extension of classical planning: if a task's preconditions hold when it is *started*, then the task is executable. An action that is executable transforms the state into a new state by applying its effects to the state in which it was executed. The effects of plan disturbance $\xi(\pi)$ are applied in the state following the execution of $\text{pred}(\xi(\pi))$.

[3]We also permit the special value ϵ for disturbances that occur before any actions.

Listing 2: "PDDL methods" for SHOP3.

```
<method> ::= (:method <method-task>
                :method-name <symbol>
                :variables <typed-list (variable)>
                :precondition <GD>
                :task-net <task-net>)

<method-task> ::= (<task-symbol> <task-arg>∗)
<task-arg> ::= <name> | [<variable> -<type>]
<task-net> ::= (<task-net-component>+)
<task-net-component> ::= (<task-symbol>
                             <method-task-arg>∗)
<method-task-arg> ::= <term>
```

3 Our SHOP3 Extensions

Our approach to plan repair is based on the existing SHOP3 planner (Goldman and Kuter 2019), with some extensions, and previous work on plan repair in UMCP-style HTN planners (Ayan et al. 2007). Two key extensions are the addition of PDDL handling to SHOP3, and the addition of an alternative search engine that can perform backjumping. We add PDDL handling for two reasons: (1) PDDL has a clearer, and easier to handle semantics than does SHOP3's language: in theory SHOP3's language may comply with PDDL semantics, but in practice, it has the full expressive power of a temporal extension of Prolog. (2) Incorporating PDDL allows us to handle benchmarks and make comparisons.

PDDL in SHOP3 Now SHOP3 can plan with both native operators and PDDL actions, and can incorporate PDDL domains in its own domains by reference. Additionally, we have developed "PDDL methods" for SHOP3.

The STRIPS dialect of SHOP3 is as follows: (1) primitives are PDDL typed STRIPS operators (i.e., equivalent to PDDL requirements of `:typing :negative-preconditions`), and (2) the method grammar is given in Listing 2. In that grammar, `<GD>` and `<typed list (variable)>` are as defined in the PDDL 2.1 grammar (Fox and Long 2003). `<task-symbol>` is the same as PDDL 2.1's `<action-symbol>`, however additionally each `<task-symbol>` is classified as either *primitive* or *complex*. `<method-task-args>` and arguments in preconditions, if variables, must be elements of the parameter list. `<method-name>` is also equivalent to `<action-symbol>`: it designates a unique method for achieving the `<task-spec>`. This is equivalent to the following limitations:

1. Typed STRIPS dialect of PDDL for the primitives;
2. Conjunctive preconditions for the methods, no SHOP3 special language features.
3. For methods, only variables in the task parameters and in the `:variables` list are scoped over the preconditions and effects (standard SHOP3 has Prolog-style scoping).
4. All tasks will be ground when added to the plan, and all task parameters (including those for complex tasks) will be ground before the preconditions are checked.

We will initially explain our approach for the simple case of the STRIPS dialect, and then will extend to "ADL SHOP3." The ADL dialect of SHOP2 uses as operators PDDL actions in the ADL dialect, and permits ADL-style quantification in method preconditions (but not functional terms or numerical fluents). So we add to the method preconditions grammar the ability to use `forall` and `exists`, and the full set of boolean connectives.

Note that neither of our SHOP dialects supports SHOP3's `:unordered` construct, which permits partial ordering in a method's task network. We are interested in doing so, but this causes issues with the semantics of plan executability that we discuss further in our conclusions.

Backjumping Another key extension to SHOP3 was to add the ability to do backjumping search. Ordinarily, if SHOP3 makes a poor decision at a state, the search process will lead to dead ends, and it must back up to that state and resume searching with a different decision. The simplest approach to "backing up" is *chronological backtracking*: undoing the most recent decision in the search and trying an alternative. However, in some problems, it is possible to determine that the most recent decision was not relevant to reaching the dead end. Backjumping (Gaschnig 1979) exploits such information by skipping over irrelevant decisions and backtracking directly to the most recent *relevant* decision, d, undoing all the decisions above d in the search stack.

4 Plan Repair

The basic idea behind our plan repair approach is very simple: when a disturbance is introduced into the plan, SHOPFIXER will find 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 un-executable) by that disturbance, and restarts the planning search from that task's immediate parent in the HTN plan (since that was the point at which 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 (see the discussion in the conclusions, Section 8). 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

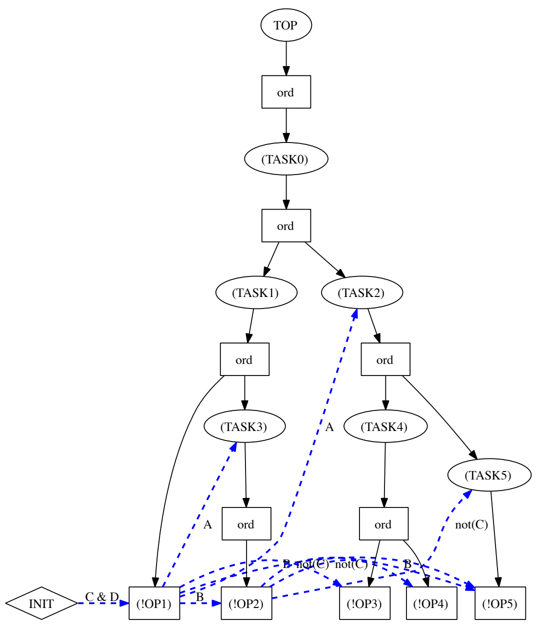


Figure 1: A high-level illustration of causal links over a task hierarchy generated by SHOP3.

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.

From this simple outline, it can be seen that the critical requirement for plan repair is to find the chronologically first clobbered task. For the STRIPS dialect of SHOP3, we will see that we can provide sound and complete detection of the first clobbered task. However, we cannot do this efficiently for the ADL dialect, because of conditional execution. For the ADL dialect, we can only provide completeness (if a is the first clobbered task in the plan, we will find it or a predecessor), and not soundness (a may appear before the first clobbered task, but no task before it is clobbered). For this reason, it follows from the soundness and completeness of SHOP3 that our plan repair algorithm is sound and complete, but it may do more work than is necessary.

SHOPFIXER finds the first clobbered task using causal links. We have enhanced SHOP3 so that whenever it inserts a task into the plan, it associates with the task causal links to its preconditions. A causal link is a triple, $\langle e, p, t\rangle$, where e is the action that established the ground literal p, and t (the "consumer") is a task that requires p in its preconditions. Figure 1 illustrates an abstract depiction of a graphical representations of causal links over a task hierarchy generated by SHOP3. SHOP3 maintains a link only for the *latest* establisher of p relative to t, as is required for soundness. The causal links are hashed with p as the key, so that if a disturbance clobbers p, SHOPFIXER can find the first consumer.

Given the simplicity of finding the first clobbered consumer from a proposition, p, the soundness and completeness of this task, depends on the soundness and completeness of causal links. Completeness is readily established, since it requires simply recording the establisher of each proposition or, in worst case, searching backwards through the state trajectory of the plan, which SHOP3 maintains (implicitly) for purposes of backtracking/backjumping.

Soundness is also immediate, with the exception of the complex tasks. For a complex task, there may be variables (in the `:variables` property of the method, see Listing 2) not bound until after the method is chosen. Those variables are effectively existentially quantified (their bindings come from a refutation of the negation of the preconditions), and SHOP3 records causal links for the *grounded* preconditions. So, for example, if we have a method like the following:

```
(:method (achieve-goals)
   :variables
     (?obj - objective ?mode - mode)
   :precond
     (communicate_image_data ?obj ?mode)
   :task-net
     (:ordered
       (communicated_image_data
          ?obj ?mode)
       (achieve-goals)))
```

which chooses an objective and a mode from the goals, and SHOP3 chooses `obj1` for `?obj` and `h_res` for `?mode`, then it will record the causal link

$$\langle e, \texttt{(communicated\_image\_data obj1 h\_res)}, t\rangle$$

Assuming that a disturbance countermands this task (i.e., deletes `(communicated_image_data obj1 h_res)`), then arguably treating t as clobbered is incorrect, because the task could be retained, and query for

```
(communicate_image_data ?obj ?mode)
```

rather than replanning its parent. However, in practice, SHOP3 binds these variables concurrently with choosing the task, so if this is unsound, it is a benign unsoundness: when replanning from t's parent, SHOP3 will first consider alternative bindings for `?obj` and `?mode`, and any siblings to the right of this task will be replanned anyway.

We have shown that finding the first clobbered task can be performed in a sound and complete way for the STRIPS dialect of SHOP3. We use this to establish the soundness and partial completeness of the ADL dialect. Recall that ADL adds logical connectives, quantification (`forall` and `exists`), and conditional effects. It is conceptually simple to account for the logical connectives and quantification, because of PDDL's finite domain of quantification:

AND handled as above;

NOT Negated ground literals can be handled identically to positive ground literals, complex negations are rewritten to drive the negations inward;

OR take the causal links from one conjunct or another;

forall over a fixed, finite domain, a conjunction.

exists over a fixed, finite domain, disjunction.

imply this is equivalent to a disjunction.

Note that the use of disjunction already compromises the soundness of finding the first clobberer, because a task with the precondition `(or p q)` that is executed in a state satisfying $p \wedge q$, will get only one causal link, for p or q, so it may be incorrectly retrieved if only one of the two is deleted by a disturbance. In practice, we have not found this unsoundness to be problematic, but it is certainly possible that it would be, particularly in a domain with existentially quantified preconditions ranging over a large set, such as "there must be an employee sitting at a monitoring station," if there are many employees, many monitoring stations, and more than one employee at a single monitoring station.

A second cause of unsoundness comes from conditional effects. Recall that conditional effects are of the form `(when p e)`, where p are the *secondary preconditions* for e relative to the operator with this effect. When SHOP3 adds a to the plan, it will add causal links for p relative to every e that is instantiated, and `(not p)` for every e not instantiated (the latter happens because of the handling of quantifiers that contain the `when` effect). In other words, instead of capturing the causal links for "every effect of a that is used in the plan," SHOPFIXER uses links for "the secondary preconditions that cause a to have *exactly this set* of effects."

We tolerate the unsoundness in order to avoid reasoning about arbitrary logic in preconditions, and chaining conditional effects through multiple stages in a plan. Trying to compute the clobbering conditions exactly would in the worst case involve solving SAT problems. Section 6 illustrates how this tradeoff plays out empirically: to date it seems benign – the savings for plan repair are still substantial, and the cost of maintaining the causal links manageable.

5 LPG Extensions

Recall that Fox, *et al.* (2006) were limited to repairing/replanning only in response to changes in the initial state, rather than changes in the middle of plan execution. We have extended their version of LPG with a preprocessor that handles disturbances in the middle of plans. Using a modified initial state I' and/or goal condition set G' along with plan π that solves an original classical planning problem $\mathcal{P} = \langle F, A, I, G\rangle$, they defined the replanning problem as $\mathcal{P}_{replan} = \langle F, A, I', G'\rangle$ and the repair problem as $\mathcal{P}_{repair} = \langle F, A, I', G', \pi\rangle$. However, just as SHOPFIXER can begin the process at an intermediate state in the plan

$$s_{div} = a_{d-1}\left(a_{d-2}\left(\ldots\left(a_1\left(I\right)\right)\ldots\right)\right)$$

where the divergence occurs just before executing action a_d, we also update the initial state in LPG to be s_{div}. Then the replanning problem is simply $\mathcal{P}_{replan} = \langle F, A, s_{div}, G\rangle$ and the repair problem is $\mathcal{P}_{repair} = \left\langle F, A, s_{div}, G, \pi_{d..|\pi|}\right\rangle$ where $\pi_{i..j}$ is the subsequence of the plan π from actions a_i to a_j, inclusive. That is, LPG begins the planning process in the state where the divergence occurs and repairs the remaining actions in the plan (without the task network information that is available to SHOPFIXER).

6 Experiments

To assess the efficiency and correctness of our approach we have conducted preliminary experiments with the "Openstacks"[4], "Rovers"[5], and "Satellite"[6] domains from the IPC. We chose these domains for their use of the ADL dialect of PDDL. Our results show a substantial savings for using plan repair over replanning from scratch in an unexpected state during plan execution.

Problems for experimentation were constructed from the IPC problems, giving a sliding scale of difficulty for each domain. For each domain, we defined a set of disturbance pseudo-actions, which involved events like shipping failures (in Openstacks), loss of samples, obstructions to line of sight (in Rovers), direction changes, decalibrations, and power loss (in Satellite). The disturbance actions had to be carefully written in order to not render a problem unsolvable, or to change the class of the problem. For the notion of class of the problem, we drew on Hoffmann's (Hoffmann 2005) analysis. For example, we avoided breaking the symmetry of the `can_traverse` relationship in the Rovers problems. This property is not formalized in any way in the domain (PDDL does not support higher-order assertions about problems), but it is present in all of the problems nevertheless. Changing the character of the problems with disturbances would render the repair and replanning less directly comparable. HTN domains for these problems, the problems themselves, disturbance actions, and raw results are available on the web (Goldman, Kuter, and Freedman 2020).

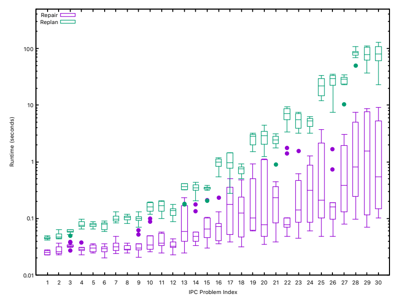
Computational Efficiency Figure 2 shows the comparison between the time to find the original plan and the time to repair the plan for SHOP3. Our experiments with SHOP3 support Fox *et al.*'s argument for plan repair over replanning, and also show the value of our replanning method for SHOP3. For each problem instance along the x-axis, the boxes mark the lower quartile q_1, median, and upper quartile q_3 of the set of times taken per run T. The extended whiskers indicate the interval of times taken per run that are within 1.5 times the interquartile range from these quartiles; that is, a line from the minimum to the maximum of $W = \{t \in T\,|q_1 - 1.5\,(q_3 - q_1) \leq t \leq q_3 + 1.5\,(q_3 - q_1)\}$. Any runs whose times are outside W are marked with a dot. The results here are for 10 runs for each problem instance of Openstacks, Rovers, and Satellite. Each run has a randomly generated deviation that clobbers the preconditions for some action in the plan suffix, rendering the plan un-executable or preventing it from achieving the goal. Note that Openstacks runtimes, unlike those for the other two domains, are plotted on a logarithmic scale, because of their high variance.

Recall that in order to make a plan repairable, SHOP3 must save much more information in the course of planning. In particular, it must build a plan tree that records causal links between actions and the tasks that consume their effects. In order to determine whether our plan repair is simply a tradeoff between spending more time in planning to save time in plan repair, we compare the runtime of generat-

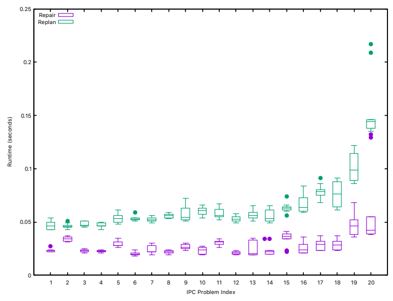
[4]`http://icaps-conference.org/ipc2008/deterministic/`

[5]`http://ipc02.icaps-conference.org/`

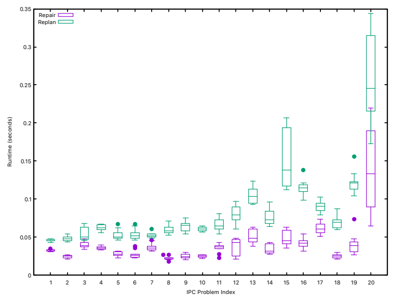
[6]`http://ipc04.icaps-conference.org/deterministic/`



(a) Openstacks problems

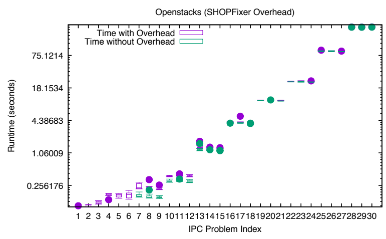


(b) Rovers problems

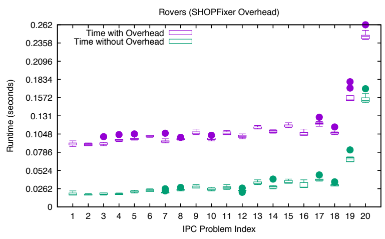


(c) Satellite problems

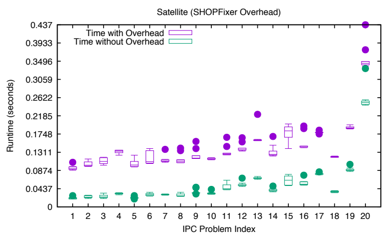
Figure 2: Comparing plan repair time to replanning, SHOP3.



(a) Openstacks problems



(b) Rovers problems



(c) Satellite problems

Figure 3: Overhead when planning for repair-ability, SHOP3.

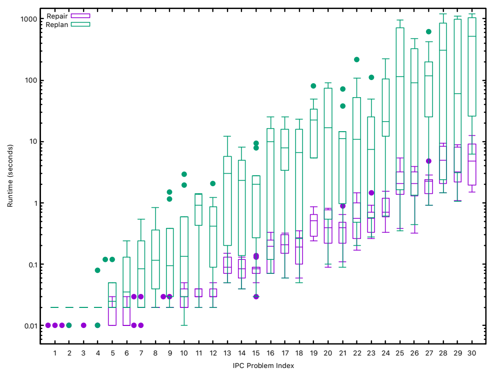


Figure 4: Comparing Openstacks repair time to replanning, LPG.

ing repairable plans with the additional information needed for replanning together with the runtime for generating plans without the additional information. Results are in Figure 3.

We ran the same problems and deviations using LPG-repair (henceforth “LPG”), with the preprocessing described above. Results on Openstacks, in Figure 4, confirm those with SHOP3: they show a high variance (Openstacks runtimes are plotted logarithmically), and show a clear advantage for plan repair, in terms of runtime. Results on Rovers (Table 1) and Satellite on the other hand, are equivocal, showing no clear advantage for repair over replanning. They exhibit a floor effect: runtimes for these problems are not significant for LPG. We have omitted the table for Satellite to save space: it is essentially identical to that for Rovers.

Plan Stability Metric Another proposed advantage of plan repair over replanning is *stability*: repaired plans will be more similar to the original plans than entirely new plans from replanning. In their paper, Fox et al. (2006) measure plan stability using *Action Distance* (AD). Briefly, this is the cardinality of the symmetric set difference of the actions in the two plans. We have argued elsewhere (Goldman and Kuter 2015) that there are a number of problems with this definition: it's insensitive to the ordering of steps in the plan, it treats `(drive truck1 src dest)` and `(drive truck2 src dest)` as just as different as `(drive truck1 src dest)` and `(navigate vaporetto1 src dst)`, *etc.*. We propose the use of *Normalized Compression Distance* (NCD) as a better alternative to AD. For an example of why we prefer NCD, see Figure 5. One can easily see that AD provides a much less stable measure of distance, so we use only NCD going forward.

Plan Stability Our experiments with SHOP3 and LPG confirm and *extend* the Fox et al. (2006)'s results: extend because we support plan upsets anywhere during execution. Our results on plan stability, measured using NCD are shown in Figure 6 for SHOP3, and 7 for LPG. These give distance between repaired and replanned plans and original plans, showing that for both planners repair improves stability.

7 Related Work

Previous extensions of the SHOP framework include HOTRiDe (Ayan et al. 2007) and SHOPLIFTER!(Kuter 2012). SHOPLIFTER augments SHOP2's HTN representations and planning capabilities with a constraint-based formalism for HTNs, inspired by UMCP (Erol, Hendler, and

Problem	Repair Time mean	Repair Time std	Replan Time mean	Replan Time std
1	0.02	0.00	0.02	0.00
2	0.02	0.00	0.02	0.00
3	0.02	0.00	0.02	0.00
4	0.02	0.00	0.02	0.00
5	0.02	0.00	0.02	0.00
6	0.02	0.00	0.02	0.00
7	0.02	0.00	0.02	0.00
8	0.02	0.00	0.02	0.00
9	0.02	0.00	0.02	0.00
10	0.02	0.00	0.02	0.00
11	0.02	0.00	0.02	0.00
12	0.02	0.00	0.02	0.00
13	0.02	0.00	0.02	0.00
14	0.02	0.00	0.02	0.00
15	0.02	0.00	0.02	0.00
16	0.02	0.00	0.02	0.00
17	0.02	0.00	0.02	0.00
18	0.03	0.00	0.03	0.00
19	0.03	0.00	0.04	0.01
20	0.05	0.01	0.04	0.01

Table 1: LPG replan and repair times for Rovers domain.

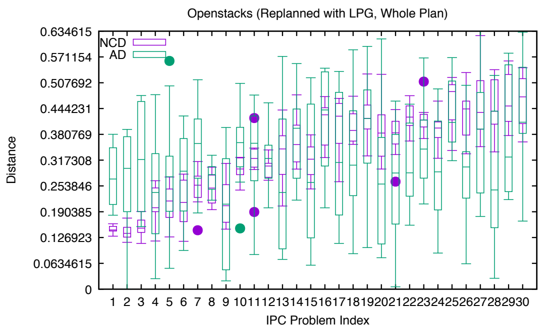


Figure 5: Comparing NCD and AD.

Nau 1994). These constraints provide the required representation conditions that need to hold during the execution of a task network as well as action post-conditions and plan goals. Neither HOTRiDe nor SHOPLIFTER provide the guarantees of correctness we give here.

Warfield et al. (2007) developed a replanning algorithm called RepairSHOP, similar to HOTRiDe. They differ in their dependency representations. RepairSHOP uses a more general and expressive data structure, a "GoalGraph." Although GoalGraphs would enable the planner to produce explanations for task dependencies and replan using those explanations, it is not clear how the two approaches compare in terms of expressive power and efficiency. Unfortunately, RepairSHOP was not available for use in our comparison experiments. Schattenberg (2009) also uses a partial-order causal link (POCL) formalism, a generalization of our totally-ordered causal links. Their repair strategies resemble ours, but do not attempt to maintain stability.

Recent work by Höller, *et al.* (2018) works from a UMCP-like POCL basis, rather than forward planning as we do. They pose the plan repair problem as a constrained HTN planning problem, by transforming the original problem description. This approach allows them to use a "stock" planner for repair, not requiring a separate repair algorithm. Most interestingly, they attempt to fully honor the constraints implied by the task networks, in a way we do not. Arguably this is more correct, but equally it could be argued that this allows only repairs with counterintuitive limitations. Their system requires that all completed actions be part of any task network constructed in repair. We show the difference between our approaches in Figure 8. Consider the simple plan shown as 8(a), and a case where after the execution of a_3 there is a disturbance that prevents executing b_1. Both systems can generate the repair in 8(b). But now consider what happens if a disturbance after a_2 makes a_3 impossible. SHOPFIXER could generate the repair in 8(c), but Höller, *et al.*'s system would regard it as incorrect, because the repaired plan tree does not include a_1 or a_2.

Bidot, Schattenberg, and Biundo (2008) present a plan repair method based on plan-modification. Their approach identifies disturbances that might break the causal dependencies in the search space of a HTN planner and produces alternatives to patch the plans. Both Bidot, Schattenberg, and Biundo and our work has been based on the ideas from (Ayan et al. 2007; Wilkins and desJardins 2001; Kambhampati and Hendler 1992). Our work uses the SHOP3 framework to generate plan repairs on the fly, so SHOPFIXER's data dependencies and repairs are incorporated in the planning algorithm: SHOPFIXER interleaves planning, plan repair, and execution using the same data structures.

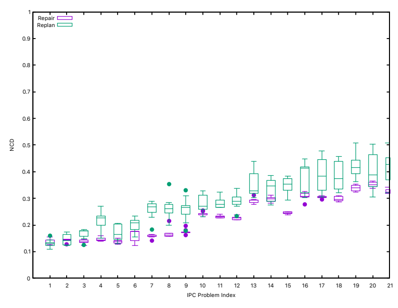
Although Bidot, Schattenberg, and Biundo's method uses the ADL-like dialect of PDDL, this seems to be limited to the primitive tasks and state representations. SHOPFIXER supports plan-repair over universally and existentially quantified expressions, increasing the HTN models that can be repaired by SHOPFIXER both theoretically and practically.

Wang and Chien describe a planning algorithm (1997) for replanning HTNs as formalized by Erol, Hendler, and Nau (1994). They extend the DPLAN algorithm (Chien et al. 1996) to replanning. Their approach is similar to HOTRiDe, but relies on the assumption that facts can be restored to their initial state when the plan fails. We did not make this assumption since it does not fit many real world domains.

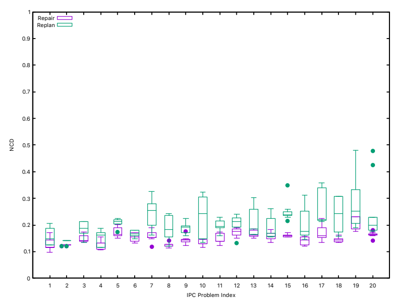
8 Conclusions

Our work addressed the issue of achieving plan stability through plan repair, as opposed to *ab initio* replanning. Our results with both SHOP3 and LPG-repair confirm, refine, and extend earlier results on repair vs. replanning from Fox et al. (2006). The one exception is that we did not universally find a computational advantage for LPG-repair over replanning: this is likely an artifact of the test domains.

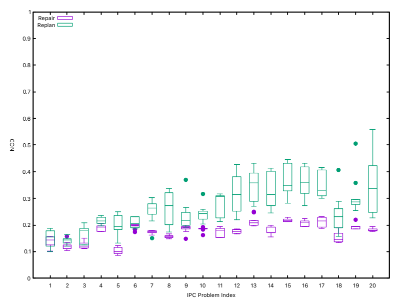
Going beyond previous work, we have provided a method for minimal-perturbation replanning for SHOP3 and shown it to be sound and complete, thus going beyond previous work in this area (Ayan et al. 2007; Kuter 2012). However, while sound and complete, SHOPFIXER is built on a method



(a) Openstacks problems

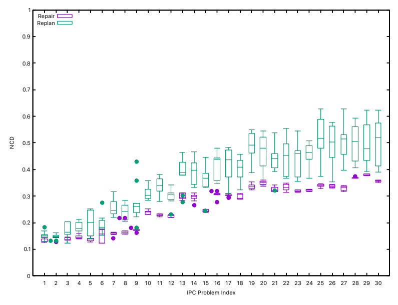


(b) Rovers problems

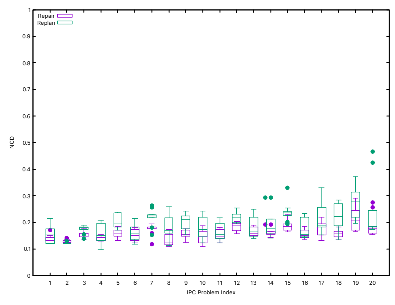


(c) Satellite problems

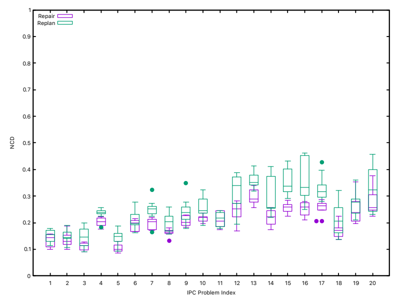
Figure 6: SHOP3 replan NCD versus repair NCD.



(a) Openstacks problems



(b) Rovers problems



(c) Satellite problems

Figure 7: LPG replan NCD versus repair NCD.

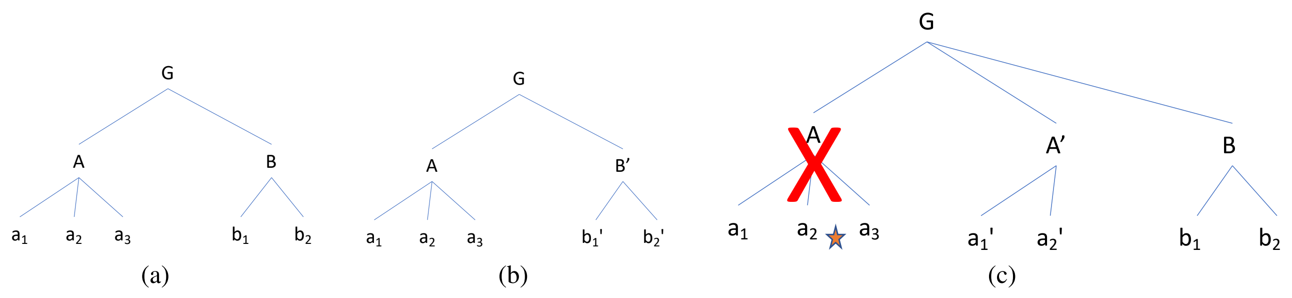


Figure 8: Höller *et al.*'s plan repair versus SHOPFIXER.

of detecting plan flaws that is complete but *unsound*, meaning that it can do extra work in some cases. We have also shown empirically that the bookkeeping overhead required by SHOPFIXER does not unduly burden planning.

In future work, we wish to extend the applicability of our techniques. SHOP3 is often used precisely because it can handle problems beyond PDDL's expressive power: the preconditions language has full Prolog expressive power, domains of quantification may not be finite, and preconditions can invoke arbitrary code. We would like to extend SHOPFIXER's expressiveness beyond the current "PDDL-like" limitations. Also, we would like to extend SHOPFIXER's stability: at present, if SHOPFIXER repairs a task T_1 generated by a method $M \rightarrow T_1 \ldots$, then the previous expansion of T_1's right siblings is lost. HOTRiDe (Ayan et al. 2007) did not have this limitation, but as mentioned earlier, used POCL planning, and is not known to be sound and complete. We are investigating *analogical replay* (Goldman et al. 2000) to retain existing plan suffixes.

A final remark: our work highlights the need for a more robust notion of "planning domain." A PDDL domain is only a way to use the same set of operators in multiple problems; it does not capture state constraints. For example, the fact that all logistics networks are fully connected, and all links are symmetric is not captured in PDDL. Hoffmann (2005) had to find these properties by empirical analysis of problem instances. Such constraints are captured only implicitly in bespoke programs that generate problems. This *substantially* complicated the process of assembling disturbance operators, and led us to rule out addressing goal changes. AI planning needs a more robust notion of domain to address issues like plan repair, planning and execution, learning, *etc.*

Acknowledgments The work reported in this paper project is sponsored by the Air Force Research Laboratory (AFRL) under contract FA8750-16-C-0182 for the Distributed Operations program. Cleared for public release, no restrictions.

References

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

Bidot, J.; Schattenberg, B.; and Biundo, S. 2008. Plan repair in hybrid planning. In *Annual Conference on Artificial Intelligence*, 169–176. Springer.

Chien, S.; Govindjee, A.; Estlin, T.; Wang, X.; and Jr., R. H. 1996. Integrating hierarchical task network and operator-based planning techniques to automate operations of communications antennas.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proc. National Conf. on Artificial Intelligence (AAAI)*.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR* 20:61–124.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan Stability: Replanning versus Plan Repair. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *ICAPS*, 212–221. AAAI.

Gaschnig, J. 1979. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. San Francisco, CA: Morgan Kaufmann.

Goldman, R. P., and Kuter, U. 2015. Measuring Plan Diversity: Pathologies in Existing Approaches and A New Plan Distance Metric. In *Proceedings of the Twenty-Ninth AAAI Conference*. AAAI Press.

Goldman, R. P., and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: The case of SHOP3. In *Roceedings of the 12th European Lisp Symposium*.

Goldman, R. P.; Haigh, K. Z.; Musliner, D. J.; and Pelican, M. 2000. MACBeth: A Multi-Agent Constraint-Based Planner. In *Working Notes of the AAAI Workshop on Constraints and AI Planning*, 11–17.

Goldman, R. P.; Kuter, U.; and Freedman, R. G. 2020. plan-repair-icaps2020-htnws. GitHub repository with test data and experimental results. {`https://github.com/shop-planner/plan-repair-icaps2020-htnws`}.

Hoffmann, J. 2005. Where ”ignoring delete lists” works: Local search topology in planning benchmarks. *JAIR* 685–758.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. HTN plan repair using unmodified planning systems. In *Proceedings of the First ICAPS Workshop on Hierarchical Planning*, 26–30.

Kambhampati, S., and Hendler, J. A. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence* 55:193–258.

Kuter, U. 2012. Dynamics of behavior and acting in dynamic environments: Forethought, reaction, and plan repair. Technical Report 2012-1, SIFT.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avia, H. 1999. SHOP: Simple Hierarchical Ordered Planner. Technical Report CS-TR-3981, University of Maryland, College Park.

Nau, D.; Au, T. C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20:379–404.

Schattenberg, B. 2009. *Hybrid Planning And Scheduling*. Ph.D. Dissertation, Ulm University, Institute of Artificial Intelligence. URN: urn:nbn:de:bsz:289-vts-68953.

Srivastava, B.; Nguyen, T. A.; Gerevini, A.; Kambhampati, S.; Do, M. B.; and Serina, I. 2007. Domain Independent Approaches for Finding Diverse Plans. In *Proceedings IJCAI*.

Wang, X., and Chien, S. 1997. Replanning using hierarchical task network and operator-based planning. In *Proc. European Conf. on Planning (ECP)*.

Warfield, I.; Hogg, C.; Lee-Urban, S.; and Munoz-Avila, H. 2007. Adaptation of hierarchical task network plans. In *FLAIRS-2007*.

Wilkins, D., and desJardins, M. 2001. A call for knowledge-based planning. *AI Magazine* 22(1):99–115.