Planning Task Shielding: Detecting and Repairing Flaws in Planning Tasks through Turning them Unsolvable

Alberto Pozanco, Pietro Totis, Marianela Morales, Daniel Borrajo

J.P. Morgan AI Research

{alberto.pozancolancho,marianela.moraleselena}@jpmorgan.com, {name.surname}@jpmorgan.com

arXiv:2604.07042v3 [cs.AI] 11 Aug 2026

Abstract

Most research in planning focuses on generating a plan to achieve a desired set of goals. However, a goal specification can also be used to encode a property that should never hold, allowing a planner to identify a trace that would reach a flawed state. In such cases, the objective may shift to modifying the planning task to ensure that the flawed state is never reached, in other words, to make the planning task unsolvable. In this paper we introduce planning task shielding: the problem of detecting and repairing flaws in planning tasks. We propose three different approaches to solve these tasks, each aiming to minimize the number of modifications to the original actions that render the planning task unsolvable.

Introduction

Classical planning is the task of finding a plan, which is a sequence of deterministic actions that, when executed from a given initial state, lead to a state where some given goals are true (Ghallab, Nau, and Traverso 2004). Most research in planning focuses on generating plans to solve the given task, assuming such a plan exists.

However, planning can also be applied in the opposite way. This approach involves formalizing the system and the security property to be verified as a planning task. If this planning task is proven to be unsolvable (Eriksson, Röger, and Helmert 2017; Ståhlberg, Francès, and Seipp 2021), it indicates that the security property is upheld within the system. Conversely, if a solution is found, the resulting plan outlines a sequence of steps or actions that could potentially falsify the security property. This approach to planning has been utilized to identify flaws in cybersecurity systems (Boddy et al. 2005; Hoffmann 2015) and cryptographic protocols (Pozanco et al. 2021). In these systems, when a flaw is detected, a domain expert reviews the plan that leads to it and manually modifies the system's dynamics (its actions) to prevent that trace from occurring, hoping this will resolve the issue. However, addressing the flaw locally may introduce new vulnerabilities elsewhere in the system, potentially resulting in a cycle that is tedious and difficult to resolve.

We illustrate these type of problems with a simple running example (Listing 1), which describes an approval workflow in PDDL (Haslum et al. 2019). Since the workflow may have been created by non-experts, it could contain errors and represents a best-effort formalization. The set of fluents is ⟨documents_submitted, application_complete, granted_approval, escalated, client_concerns, safe_client⟩. The submit_application action completes an application if all documents are submitted; direct_approval allows direct approval for completed applications, and escalation handles cases with client concerns. We may want to check if flawed states can arise, such as reaching a state where an application is both granted_approval and escalated, starting from documents_submitted and client_concerns. The plan $\pi = (\text{submit_application}, \text{escalation}, \text{direct_approval})$ achieves this, indicating the workflow is ill-defined. Addressing this requires modifying actions, such as removing escalated from the escalation action effects; or adding safe_client as a precondition to direct_approval. Selecting the best fix depends on the domain and action semantics, and care is needed, as local changes may introduce new vulnerabilities elsewhere in the workflow.

Listing 1: Approval workflow described as PDDL actions.

```
(:action submit_application
  :parameters ()
  :precondition (documents_submitted)
  :effect (and (application_complete)
               (not (documents_submitted))))

(:action direct_approval
  :parameters ()
  :precondition (application_complete)
  :effect (granted_approval))

(:action escalation
  :parameters ()
  :precondition
  (and (application_complete)
       (client_concerns))
  :effect (escalated))
```

In this paper, we propose an extension to the traditional approach of identifying flaws in systems represented as planning tasks by introducing the capability to automatically fix these flaws. Our method focuses on making the planning task unsolvable, in contrast to domain repair works (Gragera et al. 2023; Lin, Grastien, and Bercher 2023; Bercher, Sreedharan, and Vallati 2025; Gragera et al. 2025), which aim to mod-

Copyright © 2027, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

ify the planning task to make it solvable. We refer to this problem as *planning task shielding*, and formally define it as finding the minimal set of precondition additions, add effect removals, and delete effect additions to the original set of actions such that the planning task becomes unsolvable.

We then propose three different approaches to solve these tasks, each offering distinct theoretical guarantees and empirical performance. The first approach computes all loopless plans that solve the planning task, and then formulates an optimization problem to determine the minimal set of modifications required to invalidate all these plans. The second approach introduces a family of algorithms that search within the space of possible modifications to render the planning task unsolvable. The third approach conducts a static analysis of the planning task to identify actions that achieve the goals, and then modifies these actions so that the goals can no longer be achieved.

The remainder of the paper is organized as follows. We first formalize classical planning and Monte Carlo Tree Search. We then provide a formal definition of planning task shielding problems and their solutions, followed by a presentation of our three approaches for solving these tasks. Next, we evaluate the proposed algorithms on existing and novel benchmarks, highlighting the trade-offs between their theoretical guarantees and empirical performance. Finally, we conclude by discussing the main results and outlining potential directions for future research.

Background

Classical Planning

We formally define a planning task as follows:

Definition 1 (Planning task)**.** *A planning task is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{F}$ is a set of fluents, $\mathcal{A}$ is a set of actions, $\mathcal{I} \subseteq \mathcal{F}$ is an initial state, and $\mathcal{G} \subseteq \mathcal{F}$ is a goal specification.*

A state $s \subseteq \mathcal{F}$ is a set of fluents that are true at a given time. A state $s \subseteq \mathcal{F}$ is a goal state iff $\mathcal{G} \subseteq s$. Each action $a \in \mathcal{A}$ is characterized by the following components. A set of preconditions $\textsc{pre}(a)$, which are set of fluents that need to be true for the action to be applied. Add and delete effects $\textsc{add}(a)$ and $\textsc{del}(a)$, which are set of fluents that are added (or deleted) once the action is applied. We assume $\textsc{add}(a) \cap \textsc{del}(a) = \emptyset$. And finally a cost $c(a) \in \mathbb{R}$ associated with performing the action. An action a is applicable in a state s iff $\textsc{pre}(a) \subseteq s$. We define the result of applying an action in a state as $\gamma(s, a) = (s \setminus \textsc{del}(a)) \cup \textsc{add}(a)$. A sequence of actions $\pi = (a_1, \ldots, a_n)$ is applicable in a state s_0 if there are states $(s_1 \ldots, s_n)$ such that a_i is applicable in s_{i-1} and $s_i = \gamma(s_{i-1}, a_i)$. The resulting state after applying a sequence of actions is $\Gamma(s, \pi) = s_n$, and $c(\pi) = \sum_i^n c(a_i)$ denotes the cost of π. A sequence of actions is simple if it does not traverse the same state more than once. The solution to a planning task $\mathcal{P}$ is a plan, i.e., a sequence of actions π such that $\mathcal{G} \subseteq \Gamma(\mathcal{I}, \pi)$. We denote as $\Pi(\mathcal{P})$ the set of all simple solution plans to planning task $\mathcal{P}$. A plan with minimal cost is optimal. A planning task is unsolvable if there is no sequence of actions π such that $\mathcal{G} \subseteq \Gamma(\mathcal{I}, \pi)$.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) (Browne et al. 2012; Swiechowski et al. 2023) incrementally builds a search tree over large decision spaces, with states as nodes and actions as edges, and is particularly effective where exhaustive enumeration is infeasible (Kemmerling, Lütticke, and Schmitt 2024), for example in games (Finnsson and Björnsson 2008; Silver et al. 2016). Each iteration follows four steps: a tree policy selects a path to a leaf (selection); a new child is created for a chosen action (expansion); a default policy estimates the leaf's value (simulation); and ancestor statistics are updated accordingly (backpropagation). A popular tree policy is UCT (Kocsis and Szepesvári 2006), which scores action a in state s as $Q(s, a) + C \cdot \sqrt{\frac{\ln N(s)}{N(s,a)}}$, where $Q(s, a)$ is the reward estimate, $N(s)$ is the visit count for s, and $N(s, a)$ is the count for action a in s; the constant C, typically $\sqrt{2}$ when $Q \in [0, 1]$ (Swiechowski et al. 2023), controls the trade-off between exploiting high-reward actions and exploring less-visited ones. When the branching factor is large, *progressive widening* (Chaslot et al. 2008; Coulom 2007) limits expansion to a growing subset of actions, introducing a new one only when $\lfloor N(s)^\alpha \rfloor \geq |children(s)|$ for $\alpha \in (0, 1)$.

Shielding Planning Tasks

Given a system formalized as a planning task, where the goal specifies a property that should never be satisfied, our objective is to *shield* the system. This means modifying the planning task so that these modifications render it unsolvable. Although planning tasks can be made unsolvable by modifying $\mathcal{I}$, $\mathcal{G}$, or $\mathcal{A}$, we focus exclusively on the latter.

Remark 1. *Note that by limiting ourselves to modifying $\mathcal{A}$, we are unable to turn unsolvable planning tasks where $\mathcal{G} \subseteq \mathcal{I}$, i.e., those tasks where the empty plan $\pi = \emptyset$ is already a solution.*

To formalize this approach, we first define what constitutes an atomic action modification.

Definition 2 (Action Modification)**.** *Given a planning task $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, an action modification μ maps an action $a \in \mathcal{A}$ to an action $\mu(a) = a'$ obtained by adding or removing a single fluent $f \in \mathcal{F}$ in exactly one of $\textsc{pre}(a)$, $\textsc{add}(a)$, or $\textsc{del}(a)$.*

We write $\mathcal{M}^*(\mathcal{P})$ for the set of all action modifications over the actions in $\mathcal{A}$, abbreviated $\mathcal{M}^*$ when $\mathcal{P}$ is clear from context.

Remark 2. *We assume that applying several action modifications to an action a yields a single well-defined modified action a'. In particular, the final action a' does not depend on the order in which the modifications are applied.*

With these notions in place, we can now define a shielding planning task and its solution.

Definition 3 (Shielding Planning Task)**.** *A shielding planning task $\mathcal{S}$ is a tuple $\mathcal{S} = \langle \mathcal{P}, \mathcal{M} \rangle$, where $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ is a planning task and $\mathcal{M} \subseteq \mathcal{M}^*$ is a set of allowed modifications.*

Definition 4 (Shielding Solution)**.** *A solution to a shielding planning task $\mathcal{S} = \langle \mathcal{P}, \mathcal{M} \rangle$ is a set of action modifications $M \subseteq \mathcal{M}$ that induces a new set of actions $\mathcal{A}'$, such that the resulting planning task $\mathcal{P}_\mathcal{S} = \langle \mathcal{F}, \mathcal{A}', \mathcal{I}, \mathcal{G} \rangle$ is unsolvable.*

Although trivial modifications are possible, our aim is to identify the minimal set of action modifications that render the planning task unsolvable. We formalize this notion of optimality as follows:

Definition 5 (Shielding Solution Optimality)**.** *A Shielding Solution $\mathcal{A}'$ obtained with modifications M is optimal iff there does not exist another solution $\mathcal{A}''$ obtained with modifications M' and $|M'| < |M|$.*

In order to compute optimal (or high quality) solutions for a shielding planning task, we do not need to reason about all the possible ways in which an action can be modified. In particular, we can restrict ourselves to the set of action modifications that reduce the number of plans that solve the original planning task.

Proposition 1 (Monotonic Decrease of Solution Plans)**.** *Let $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ be a planning task and let $\mathcal{P}' = \langle \mathcal{F}, \mathcal{A}', \mathcal{I}, \mathcal{G} \rangle$ be obtained from $\mathcal{P}$ by modifying actions only by:*

- *adding preconditions, $\textsc{pre}(a) \subseteq \textsc{pre}(a')$*
- *removing add effects, $\textsc{add}(a') \subseteq \textsc{add}(a)$*
- *adding delete effects, $\textsc{del}(a) \subseteq \textsc{del}(a')$*

Then, after applying any modification, the number of plans that solve the task decreases monotonically, i.e., $|\Pi(\mathcal{P}')| \leq |\Pi(\mathcal{P})|$.

Proof. Let $s \subseteq \mathcal{F}$ be any state, and let $a' \in \mathcal{A}'$ correspond to $a \in \mathcal{A}$. Suppose a' is applicable in s in $\mathcal{P}'$. Then, by definition, $\textsc{pre}(a') \subseteq s$. Since $\textsc{pre}(a) \subseteq \textsc{pre}(a')$, it follows that $\textsc{pre}(a) \subseteq s$, so a is also applicable in s in $\mathcal{P}$. Thus, any action applicable in $\mathcal{P}'$ at a state is also applicable in $\mathcal{P}$ at the same state. Conversely, suppose a is applicable in s in $\mathcal{P}$ (i.e., $\textsc{pre}(a) \subseteq s$), if there exists $p \in \textsc{pre}(a') \setminus \textsc{pre}(a)$ with $p \notin s$, then a' is not applicable in s in $\mathcal{P}'$. Therefore, the set of applicable actions in any state in $\mathcal{P}'$ is a subset of those in $\mathcal{P}$.

For any state s where a' is applicable, the resulting state after applying a' in $\mathcal{P}'$ is $s' = (s \setminus \textsc{del}(a')) \cup \textsc{add}(a')$. Since $\textsc{add}(a') \subseteq \textsc{add}(a)$ and $\textsc{del}(a) \subseteq \textsc{del}(a')$, it follows that $s \setminus \textsc{del}(a') \subseteq s \setminus \textsc{del}(a)$ and $(s \setminus \textsc{del}(a')) \cup \textsc{add}(a') \subseteq (s \setminus \textsc{del}(a)) \cup \textsc{add}(a)$. Thus, the state reached by applying a' in $\mathcal{P}'$ is a subset of the state reached by applying a in $\mathcal{P}$.

Consider any plan $\pi' = (a'_1, \ldots, a'_n)$ that solves $\mathcal{P}'$. For each i, let a_i be the corresponding action in $\mathcal{A}$. By the above, $\pi = (a_1, \ldots, a_n)$ is also executable in $\mathcal{P}$ from $\mathcal{I}$, and the sequence of states reached in $\mathcal{P}$ contains those reached in $\mathcal{P}'$. Since the goal is the same, if π' reaches $\mathcal{G}$ in $\mathcal{P}'$, then π also reaches $\mathcal{G}$ in $\mathcal{P}$.

Therefore, every plan that solves $\mathcal{P}'$ also solves $\mathcal{P}$, i.e., $\Pi(\mathcal{P}') \subseteq \Pi(\mathcal{P})$, and thus $|\Pi(\mathcal{P}')| \leq |\Pi(\mathcal{P})|$. □

In addition to restricting the set of modifications, we can also limit the subset of actions that need to be modified to render $\mathcal{P}$ unsolvable. Let $\mathcal{A}^\Pi = \{a \mid a \in \pi, \forall \pi \in \Pi(\mathcal{P})\}$ denote the set of actions that appear in at least one plan solving the original task.

Remark 3. *It suffices to consider modifications to actions in $\mathcal{A}^\Pi$ (and not the remaining actions in $\mathcal{A}$) to render $\mathcal{P}$ unsolvable.*

Next, we present our three different approaches to solve shielding planning tasks.

ALLMIN

ALLMIN follows a two-step process: (1) it computes all the simple (loopless) plans that can solve the original planning task, $\Pi(\mathcal{P})$; and (2) it determines the set of minimal modifications to the original actions $\mathcal{A}$ that would block the execution of all the plans in $\Pi(\mathcal{P})$.

To compute the set of plans that solve the original planning task, we can utilize any planner capable of generating not just a single plan, but a set of plans for a given task (Katz et al. 2018; Speck, Mattmüller, and Nebel 2020; Speck et al. 2025). We will provide further details about the specific tool used in our experimental setup.

We formulate the task of computing the minimal modifications to the original actions that would block the execution of all the plans as a Mixed-Integer Linear Program (MILP). The MILP receives as input the planning task $\mathcal{P}$, and the set of plans that solve it $\Pi(\mathcal{P})$. We reduce the number of variables and constraints needed by leveraging Proposition 1 and Remark 3. We add a fake action a^g to $\mathcal{A}$, which represents the achievement of the goals $\mathcal{G}$. $\textsc{pre}(a^g) = \mathcal{G}$ and $\textsc{add} = \textsc{del} = \emptyset$. We append this action to each plan $\pi \in \Pi(\mathcal{P})$. We have the following set of decision variables:

- $\mathrm{pre}_{a,f} \in \{0, 1\}$: 1 if fluent f is added as a precondition to action a.
- $\mathrm{add}_{a,f} \in \{0, 1\}$: 1 if fluent f is removed from the add effects of action a.
- $\mathrm{del}_{a,f} \in \{0, 1\}$: 1 if fluent f is added to the delete effects of action a.
- $s_{\pi,i,f} \in \{0, 1\}$: 1 if fluent f holds after step i in plan π.
- $\mathrm{enabled}_{\pi,i} \in \{0, 1\}$: 1 if the action at step i in plan p is executable.
- $\mathrm{pre_unsat}_{\pi,i,f} \in \{0, 1\}$: 1 if precondition f is present and not satisfied at step i in plan π.

The objective is to minimize the total number of action modifications, specifically the addition of new preconditions, removal of add effects, and addition of delete effects:

$$\min \sum_{a \in \mathcal{A}^\Pi,\, f \in \mathcal{F} \setminus \mathrm{pre}(a)} \mathrm{pre}_{a,f} \tag{1}$$

$$+ \sum_{a \in \mathcal{A}^\Pi,\, f \in \mathrm{add}(a)} \mathrm{add}_{a,f} \tag{2}$$

$$+ \sum_{a \in \mathcal{A}^\Pi,\, f \in \mathcal{F} \setminus \{\mathrm{del}(a) \cup \mathrm{add}(a)\}} \mathrm{del}_{a,f} \tag{3}$$

The constraints ensure the correct propagation of fluents, satisfaction of preconditions, and blocking of plans:

1. **Initial State:** The initial value of each fluent for each plan is set according to the initial state $\mathcal{I}$. We include the following constraints for each $\pi \in \Pi(\mathcal{P})$ and $f \in \mathcal{F}$:

$$s_{\pi,0,f} = 1 \quad \text{if} \quad f \in \mathcal{I} \tag{4}$$

$$s_{\pi,0,f} = 0 \quad \text{if} \quad f \notin \mathcal{I} \tag{5}$$

2. **Precondition Satisfaction:** For each action in each plan, the variable $\text{pre_unsat}_{\pi,i,f}$ captures whether precondition f of action a_i is unsatisfied, considering both original and newly added preconditions. We include the following constraints for each $\pi \in \Pi(\mathcal{P})$, $a_i \in \pi$, and $f \in \mathcal{F}$:
   If $f \in \textsc{pre}(a)$:

$$\text{pre_unsat}_{\pi,i+1,f} = 1 - s_{\pi,i,f} \tag{6}$$

   If $f \notin \textsc{pre}(a)$:

$$\text{pre_unsat}_{\pi,i+1,f} \geq \text{pre}_{a,f} - s_{\pi,i,f} \tag{7}$$

$$\text{pre_unsat}_{\pi,i+1,f} \leq \text{pre}_{a,f} \tag{8}$$

$$\text{pre_unsat}_{\pi,i+1,f} \leq 1 - s_{\pi,i,f} \tag{9}$$

3. **Action Enabledness:** An action is enabled if all its preconditions are satisfied. We include the following constraints for each $\pi \in \Pi(\mathcal{P})$ and $a_i \in \pi$:

$$\text{enabled}_{\pi,i+1} \geq 1 - \sum_{f \in \mathcal{F}} \text{pre_unsat}_{\pi,i,f} \tag{10}$$

4. **State Propagation:** The fluents are updated according to the effects of the actions and the modifications. We include the following constraints for each $\pi \in \Pi(\mathcal{P})$, $a_i \in \pi$, and $f \in \mathcal{F}$:
   If $f \in \textsc{add}(a)$:

$$s_{\pi,i+1,f} \geq 1 - \text{add}_{a,f} \tag{11}$$

$$s_{\pi,i+1,f} - s_{\pi,i,f} \geq 1 - \text{add}_{a,f} \tag{12}$$

$$s_{\pi,i,f} - s_{\pi,i+1,f} \geq 1 - \text{add}_{a,f} \tag{13}$$

   Else If $f \in \textsc{del}(a)$:

$$s_{\pi,i+1,f} = 0 \tag{14}$$

   Else:

$$s_{\pi,i+1,f} \leq 1 - \text{del}_{a,f} \tag{15}$$

$$s_{\pi,i+1,f} - s_{\pi,i,f} \leq 1 - \text{del}_{a,f} \tag{16}$$

$$s_{\pi,i,f} - s_{\pi,i+1,f} \leq 1 - \text{del}_{a,f} \tag{17}$$

$$s_{\pi,i+1,f} - s_{\pi,i,f} \leq 0 \tag{18}$$

$$s_{\pi,i,f} - s_{\pi,i+1,f} \leq 0 \tag{19}$$

5. **Plan Blocking:** To ensure every plan is blocked, at least one action in each plan must not be enabled. We include the following constraints for each $\pi \in \Pi(\mathcal{P})$:

$$\sum_{i=1}^{|\pi|} \text{enabled}_{\pi,i} \leq |\pi| - 1 \tag{20}$$

6. **Goal Persistence**: We impose the following restriction to enforce that the goal is not modified, i.e., the preconditions of a^g cannot be modified.

$$\sum_{f \in \mathcal{F} \setminus \mathcal{G}} \text{pre}_{a^g,f} = 0 \tag{21}$$

This MILP identifies the minimal set of action modifications needed to block all plans. By incorporating new preconditions ($\text{pre}_{a,f} = 1$), adding delete effects ($\text{del}_{a,f} = 1$), and removing specified add effects ($\text{add}_{a,f} = 1$), the resulting set of actions $\mathcal{A}'$ ensures that the original planning task becomes unsolvable. As a result, $\mathcal{A}'$ serves as a shielding solution for the original planning task $\mathcal{P}$.

Search in the Space of Modifications

Our second approach frames the shielding problem as a search over the space of action modifications. A *search state* is a set of action modifications applied cumulatively to $\mathcal{A}$, yielding a modified action set $\mathcal{A}'$. A *search action* is a single atomic modification from Definition 2, and the goal is to reach a state where the resulting planning task is unsolvable. Even for small benchmarks, the pool of candidate modifications is large: every action contributes one candidate per fluent in each of its three components, and grounded domains can have many actions and fluents. Exhaustive enumeration is therefore infeasible: we reduce the effective branching factor through a shared action-filtering mechanism inspired by counterexample-guided search (Seipp and Helmert 2018). At each state, the planner is called on the current modified domain and if the task is unsolvable, the search has found a solution. Otherwise, the returned plan is a counterexample: it witnesses that the domain is still solvable and identifies precisely which actions enable it. The available modifications for the next step are therefore filtered to those that (a) target an action appearing in the current plan, and (b) actually invalidate that plan when simulated. This reduces the effective branching factor substantially and focuses the search on modifications that are structurally relevant.

Breadth-first search. BFS exhaustively tests all states reachable with exactly k modifications before considering any state requiring $k+1$. This guarantees that the first solution found uses the minimum number of modifications, i.e., BFS returns an optimal shielding solution in the sense of Definition 5. The cost is a branching factor that grows combinatorially with depth, making BFS impractical beyond the first few levels on larger domains.

Depth-first search. DFS follows a single branch to the specified maximum depth before backtracking. It typically reaches a solution faster than BFS when one exists at moderate depth, but offers no optimality guarantee. Moreover, for some domains the solution may be at a higher depth than the given

maximum. More fundamentally, DFS commits fully to the choices made along its current branch: without any signal to distinguish which modifications are likely to render the task unsolvable, it may spend most of its budget exploring an unproductive region of the space and only redirect its effort after exhausting it entirely.

Monte Carlo Tree Search. MCTS addresses the lack of guidance in DFS by using the simulation score to direct exploration toward promising regions of the search space. We use UCT with progressive widening: α controls the trade-off between depth and breadth, a lower value keeps the tree narrow and deep, which is effective when solutions require combining several modifications, while a higher value widens the tree earlier, favouring search over combinations of fewer modifications.

The simulation step is the planner call and the counterexample-guided filtering described above, which determines the available modifications for future expansions. Unsolvable nodes receive a score of 1. For solvable nodes, the simulation score encodes the hypothesis that longer plans and longer solve times signal a domain that is harder to solve, and thus closer to becoming unsolvable. We make no claim that this score is a reliable predictor of unsolvability: there is no domain-independent proxy for how close a modified domain is to becoming unsolvable. The score is a heuristic, computed for free from information the planner already returns, and its potential value lies in providing *some* signal to UCT rather than none. On domains with many solutions, solve times at shallow depths are nearly identical; plan length provides a more discriminating signal and is therefore the primary component, with solve time acting as a tiebreaker within the same length bucket. Both are min-max normalised across all nodes seen so far:

$$s_l = \frac{l - l_{\min}}{l_{\max} - l_{\min}}, \qquad s_t = \frac{t - t_{\min}}{t_{\max} - t_{\min}}, \tag{22}$$

where l and t are the plan length and solve time of the node. Formally, letting $b_l = \min(9,\ \lfloor s_l \cdot 10 \rfloor)$, the base score is:

$$score = \frac{b_l}{10} + s_t \cdot 0.09. \tag{23}$$

This ensures plan length dominates: no amount of solve-time difference can move a node across a length bucket boundary. When a new extreme value updates $l_{\min}$, $l_{\max}$, $t_{\min}$, or $t_{\max}$, all nodes in the tree are rescored to maintain a consistent normalisation. In practice, the search explores at most a few hundred nodes, so the cost of rescoring is negligible compared to individual planner calls.

Action Goal Achievers Analysis

Our third and final approach for solving shielding planning tasks is based on analyzing *action goal achievers*.

The approach is described in Algorithm 1. It first considers only those goal fluents that are not already true in the initial state, i.e., $\mathcal{G} \setminus \mathcal{I}$. For each $g \in \mathcal{G} \setminus \mathcal{I}$, we calculate the set of *achievers*, i.e., the actions where g appears in the add effects. The algorithm then selects the g with the smallest number of action achievers and repairs the task by deleting exactly the corresponding add effects that could achieve g, leaving all other aspects of each action unchanged.

Algorithm 1: Algorithm AGA

Input: Task $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$

$m \leftarrow +\infty$
for all $g \in \mathcal{G} \setminus \mathcal{I}$ **do**
  $A \leftarrow \{a \in \mathcal{A} \mid g \in \text{ADD}(a)\}$
  if $|A| < m$ **then**
    $g^\star \leftarrow g$; $A^\star \leftarrow A$; $m \leftarrow |A|$
  end if
end for
for all $a \in A^\star$ **do**
  $\text{ADD}(a) \leftarrow \text{ADD}(a) \setminus \{g^\star\}$
end for
return $\mathcal{P}'$

Intuitively, if we prevent the achievement of at least one goal $g \in \mathcal{G}$, then the entire goal $\mathcal{G}$ becomes unreachable, rendering the modified task unsolvable. Although this greedy selection of g may not be optimal with respect to minimizing the number of modifications, it enables us to make the planning task unsolvable without performing a search, unlike previous approaches.

Evaluation

Experimental Setting

Benchmark. We use two benchmarks to evaluate ALLMIN. In the first one, we use a synthetic benchmark (SYNTH) where we generate planning tasks in the form of a graph where we control: the number of plans $(8, 16, 32, 64)$, their maximum $(4, 8, 16, 32)$ and minimum $(2, 4, 6, 8)$ length, and the percentage of plans that share some edges with other plans (0.4). The numbers of fluents, actions, and states range from a few dozen in the smaller instances with 8 plans to a few thousand in the larger instances with 64 plans. We generate 10 random problems for each of the 4 combinations, giving us a total of 40 problems of increasing complexity. The reason we generated this synthetic benchmark is threefold. First, it allows us to control the number of plans that solve the planning task, which is not possible when working with existing tasks or using generators for known domains (Torralba, Seipp, and Sievers 2021). Second, tasks in current benchmarks are typically designed to be challenging for planners, often resulting in tasks that are too large to serve as meaningful shielding tasks. Finally, in practice, we would expect systems to have only a few ways of reaching a flawed state, rather than the hundreds of thousands of plans typically found in most planning tasks from existing benchmarks (Speck, Mattmüller, and Nebel 2020). For the second benchmark, we chose the 10 smallest problems from the following domains in the Fast Downward (Helmert 2006) benchmark collection[1]: BLOCKSWORLD, ROVERS, SATELLITE, TPP and DEPOT. By selecting this concise yet varied set of standard planning tasks, our goal is to demonstrate how the three

[1] https://github.com/aibasel/downward-benchmarks

approaches can be applied to transform unsolvable problems from domains that are well-known within the community.

Approaches and Metrics. We evaluate ALLMIN, BFS, DFS, MCTS and AGA on the benchmarks described above. We use the SYMK planner (Speck, Mattmüller, and Nebel 2020) to compute all simple plans that solve a given planning task (von Tschammer, Mattmüller, and Speck 2022). Given that computing all simple plans is not feasible for many tasks in the benchmark, we limit SYMK to output 10000 plans for ALLMIN, and we also evaluate two variants: ALLMIN^{10} and ALLMIN^{100}. Instead of computing all simple plans, these variants compute respectively 10 and 100 plans that solve the task. Problems with more plans than the SYMK limit lose the effectiveness and optimality guarantees of the modifications computed by the MILP, but may be sufficient to render the task unsolvable nonetheless. We then solve the resulting MILPs using the HiGHS solver (Hall 2019) to determine the minimal modifications required to make the planning task unsolvable. We also test DFS with different maximum depth (modifications) $d \in \{5, 10, 15\}$, denoted with DFS^d. We test MCTS with progressive widening with $\alpha = 0.2$. For each algorithm, we evaluate:

1. Completed (C): the number of instances returning a set of modifications M within the time and memory bounds.
2. Success (S): the number of instances returning a set of modifications M that turn the planning task unsolvable.
3. Time (s): the total execution time (in seconds) required to generate a solution.
4. $|M|$: the number of modifications in the solution

We validate that the suggested changes turn the planning task unsolvable by calling SYMK again and verifying there are no plans that solve the reformulated task $\mathcal{P}'$.

Reproducibility. Experiments were run on an 8-core, 2.8GHz CPU machine with 32GB RAM, with a time limit of 1800s for each shielding task.

Results

Figure 1 summarizes the number of successful modifications for each method. AGA succeeds for all 90 benchmarks and is also the fastest method. However, AGA is suboptimal and uses on average a higher number of modifications in all benchmarks but BLOCKSWORLD and SATELLITE (Figure 2). On synthetic benchmarks, AGA returns an average of 47.8% more modifications (w.r.t. the optimal ALLMIN), and up to 9 modifications more on the planning benchmarks w.r.t. the subset solved by the optimal BFS.

The second best method is ALLMIN^{100} with 49 successes out of 90, representing a balanced compromise between ALLMIN^{10}, which returns insufficient modifications in 50 instances, and ALLMIN, which runs out of memory in 43 instances. The tradeoff is clear: enumerating more plans gives more information to the MILP to find a set of effective modifications, at a cost of more memory and time resources.

In the majority of the benchmarks, 10 plans are not representative enough to find a set of modifications that invalidates all possible plans, thus ALLMIN^{10} struggles with rendering the whole task unsolvable. This behavior is evident in the SYNTHETIC benchmarks, which admit many independent paths to the goal. While ALLMIN^{10} computes valid shielding solutions for all tasks in the set with 8 plans (SYNTH4), it does not succeed in any of the other synthetic tasks. The reason is that the MILP returns the minimum number of modifications for the 10 computed plans, invalidating those plans but potentially leaving other valid solutions unaffected.

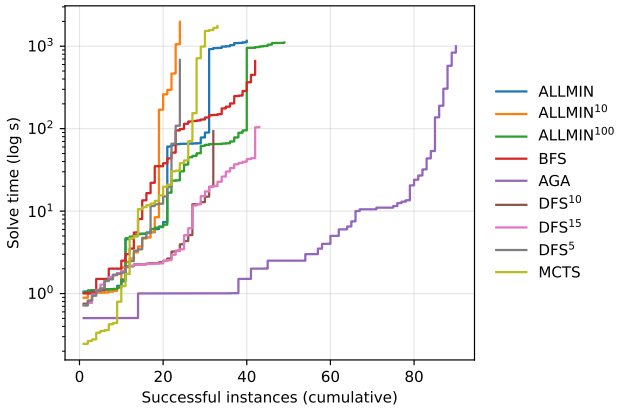


Figure 1: AGA succeeds on all instances with faster runtime, ALLMIN^{100} and DFS^{15} follow.

On the other hand, ALLMIN guarantees to return a (minimal) set of modifications invalidating all plans, but SYMK often cannot complete this enumeration within the memory and time boundaries. We analyzed how the execution time is distributed between the two main components of ALLMIN: generating all plans that solve the task using SYMK, and solving the MILP to determine the necessary modifications. The execution time of ALLMIN increases exponentially as the size of the planning task and the number of plans that solve it ($|\Pi(\mathcal{P})|$) grow, rising from a few seconds for smaller tasks to hundreds of seconds for larger ones (runtime details are available in the supplementary material).

The planning domains proved to be difficult for ALLMIN and its variants, with ALLMIN^{10} alone scoring some successes in SATELLITE and DEPOT and together ALLMIN^{100} solving most of BLOCKSWORLD. For DEPOT and TPP the bottleneck is SYMK, as ALLMIN reaches the time limit before concluding the collection of the given number of plans. On the other hand, in SATELLITE, ROVERS, BLOCKSWORLD and the larger SYNTH benchmarks, the time dedicated to solving the MILP dominates the SYMK runtime. This suggests that, for most benchmarks, the planning tasks are easier to solve than the optimization problem, which becomes increasingly complex as the number of actions, fluents, and plans grows, leading to a larger number of variables and constraints. We observe that computing a subset of all plans often enables ALLMIN^{10} and ALLMIN^{100} to achieve strong empirical performance, although there is no guarantee regarding the success of the returned modifications. ALLMIN^{100} thus balances the amount of information collected about the plans and the time spent to compute it.

Among the search methods, DFS^{15} solves the most in-

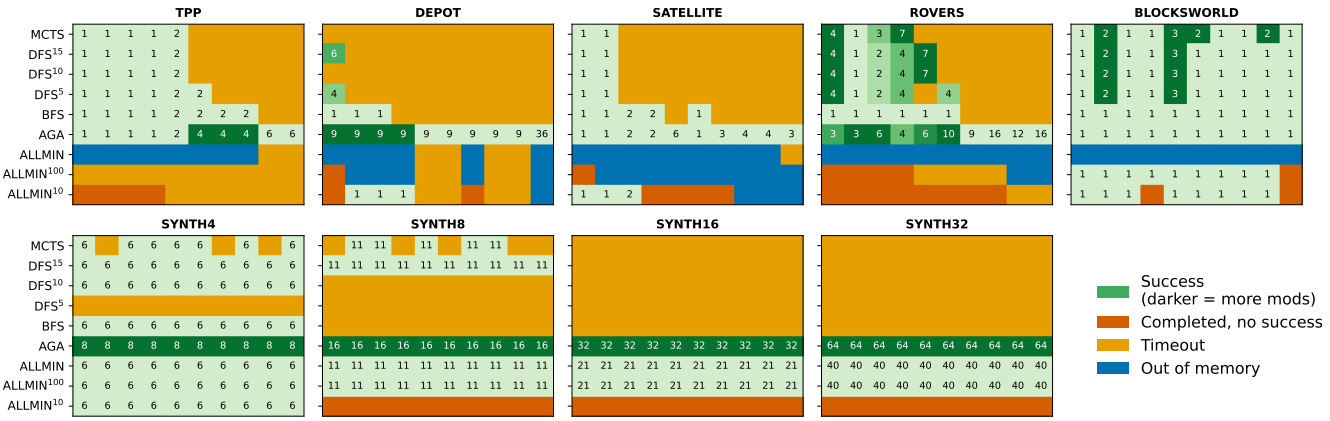


Figure 2: Experimental results on the 90 benchmark problems. Each column denotes one of the 10 problems for each domain.

stances with 43 successful modifications, followed by BFS at 42. While BFS is optimal, DFS is faster: DFS finds a successful set of modifications in 36.4% less time on average, but with an average of 79.2% more modifications. MCTS offers no advantage over DFS and BFS in either success rate or number of modifications, showing that plan length and solving time do not represent a reliable signal to guide the exploration of the action modifications.

With respect to the number of modifications, we observe two main groups. For the standard planning benchmarks, BFS is more reliable than ALLMIN at finding an optimal solution, as in many cases a single modification renders the planning task unsolvable ($|M| = 1$). This occurs because, in many of these tasks, removing a landmark fact or adding preconditions that are never achievable in certain states can make the planning task unsolvable with just one change. In contrast, the SYNTH benchmarks are more challenging, as they are designed with several independent paths leading to the goal and fewer bottlenecks. Starting from SYNTH8, the number of modifications required (11) is sufficiently large for BFS to run out of time exploring all possible combinations of modifications up to $|M| = 11$, while DFS^{15} and MCTS can still solve many SYNTH8 problems. However, SYNTH16 and SYNTH32 require a number of modifications beyond reach for the search methods, while SYMK can complete the plan enumeration quickly, allowing ALLMIN to compute an optimal set of modifications for all SYNTH problems. $|M|$ is always lower than the number of plans, indicating that ALLMIN effectively identifies actions shared across multiple plans and modifies them so that several plans become invalid simultaneously.

We also inspected and compared some of the modifications proposed by the algorithms. We noted some modifications that exploit the possibility of rendering some actions inapplicable by adding as preconditions ground facts that can never be satisfied in the domain. For example, in BLOCKSWORLD, adding the fluent $on(d, d)$ to any action precondition precludes the use of the action in any plan. This type of modifications can be prevented by restricting the set $\mathcal{M}$ of allowed modifications. Another example of modification that stretches the domain semantics is the addition of a fluent f as precondition of an action a that has the same f as effect: when a is the only action that can add f, the precondition can never be satisfied and a can never be used. For example, in SATELLITE, the action $a = calibrate_satellite0_instrument0_groundstation2$ has as effect $f = calibrated(instrument0)$ and the modification proposed by MCTS is to add f as a precondition of a. Note that this is equivalent to removing f from the effects of a, which is what BFS and DFS return in this specific instance.

Conclusions and Future Work

In this paper, we introduce planning task shielding: the problem of identifying the plans that lead to a flawed state in the original planning task and then automatically making the task unsolvable by minimally modifying the original set of actions. We formalize this problem and show that it can be addressed by considering only a subset of possible modifications to the original actions: adding preconditions, removing add effects, and adding delete effects. We then present three different approaches to solve these tasks, each aiming to minimize the number of modifications to the original actions that render the planning task unsolvable. Our evaluation demonstrates that our proposed approaches can effectively render planning tasks unsolvable, thereby shielding the system and preventing the existence of plans that reach flawed states.

Our current problem formulation consider granular modifications at the grounded action level. As next steps, we want to explore the lifted version of the problem, where we reason about modifying action schemes rather than grounded actions. We would also like to consider additional objectives, such as minimizing the number of actions to which modifications are applied, or minimizing the number of fluents used in the modifications.

Disclaimer

This paper was prepared for informational purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co. and its affiliates ("JP Morgan") and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all

liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful. © 2026 JPMorgan Chase & Co. All rights reserved

References

Bercher, P.; Sreedharan, S.; and Vallati, M. 2025. A Survey on Model Repair in AI Planning. In *34th International Joint Conference on Artificial Intelligence*. IJCAI Organization.

Boddy, M. S.; Gohde, J.; Haigh, T.; and Harp, S. A. 2005. Course of Action Generation for Cyber Security Using Classical Planning. In *ICAPS*, 12–21.

Browne, C.; Powley, E. J.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Liebana, D. P.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intell. AI Games*, 4(1): 1–43.

Chaslot, G. M.; Winands, M. H. M.; Herik, H. J. v. d.; Uiterwijk, J. W. H. M.; and Bouzy, B. 2008. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 04(03): 343–357.

Coulom, R. 2007. Computing "Elo Ratings" of Move Patterns in the Game of Go. *J. Int. Comput. Games Assoc.*, 30(4): 198–208.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability certificates for classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 27, 88–97.

Finnsson, H.; and Björnsson, Y. 2008. Simulation-Based Approach to General Game Playing. In *Proceedings of AAAI 2008, Chicago, Illinois, USA*, 259–264. AAAI Press.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: theory and practice*. Elsevier.

Gragera, A.; Fuentetaja, R.; García-Olaya, Á.; and Fernández, F. 2023. A planning approach to repair domains with incomplete action effects. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 33, 153–161.

Gragera, A.; Fuentetaja, R.; Olaya, Á. G.; and Fernández, F. 2025. On the gains from using action observations in domain repair. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 35, 343–347.

Hall, J. A. J. 2019. HiGHS. https://www.highs.dev. Accessed: 17/02/2022.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; Muise, C.; Brachman, R.; Rossi, F.; and Stone, P. 2019. *An introduction to the planning domain definition language*, volume 13. Springer.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Hoffmann, J. 2015. Simulated penetration testing: From" dijkstra" to" turing test++". In *Proceedings of the international conference on automated planning and scheduling*, volume 25, 364–372.

Katz, M.; Sohrabi, S.; Udrea, O.; and Winterer, D. 2018. A novel iterative approach to top-k planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, 132–140.

Kemmerling, M.; Lütticke, D.; and Schmitt, R. H. 2024. Beyond games: a systematic review of neural Monte Carlo tree search applications. *Appl. Intell.*, 54(11-12): 1020–1046.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *Proceedings of ECML 2006, Berlin, Germany*, volume 4212 of *Lecture Notes in Computer Science*, 282–293. Springer.

Lin, S.; Grastien, A.; and Bercher, P. 2023. Towards automated modeling assistance: An efficient approach for repairing flawed planning domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 12022–12031.

Pozanco, A.; Polychroniadou, A.; Magazzeni, D.; and Borrajo, D. 2021. Proving Security of Cryptographic Protocols using Automated Planning. *FinPlan 2021*, 38.

Seipp, J.; and Helmert, M. 2018. Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62: 535–577.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T. P.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nat.*, 529(7587): 484–489.

Speck, D.; Hecher, M.; Gnad, D.; Fichte, J. K.; and Corrêa, A. B. 2025. Counting and reasoning with plans. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 26688–26696.

Speck, D.; Mattmüller, R.; and Nebel, B. 2020. Symbolic Top-k Planning. In Conitzer, V.; and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*, 9967–9974. AAAI Press.

Ståhlberg, S.; Francès, G.; and Seipp, J. 2021. Learning generalized unsolvability heuristics for classical planning. In *The Thirtieth International Joint Conference on Artificial Intelligence, Montreal, 19-27 August 2021*, 4175–4181. International Joint Conferences on Artifical Intelligence (IJCAI).

Swiechowski, M.; Godlewski, K.; Sawicki, B.; and Mandziuk, J. 2023. Monte Carlo Tree Search: a review of recent modifications and applications. *Artif. Intell. Rev.*, 56(3): 2497–2562.

Torralba, A.; Seipp, J.; and Sievers, S. 2021. Automatic instance generation for classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 376–384.

von Tschammer, J.; Mattmüller, R.; and Speck, D. 2022. Loopless Top-K Planning. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 380–384. AAAI Press.

Supplementary Material

Plots

Figure 3 plots the average runtime breakdown for ALLMIN between the time dedicated to generating the given number of valid plans using SYMK, and solving the MILP to determine the necessary modifications. Bars with higher transparency denote the runs completed but not successful, i.e. returning a set of modifications $|M|$ such that the modified problem remains solvable. Bars exceeding the timeout are due to SYMK returning a plan after the given timeout expired.

Figure 4 shows the distribution of the number of modifications across methods and domains: AGA on average returns a higher number of modifications across all benchmarks, while ALLMIN and BFS are optimal.

Figure 5 shows the average runtimes for each method on the successful problems.

Note that both the mean number of modifications and mean runtime on some domain may not be extrapolated from the same set of problems.

Benchmarks statistics

Table 1 presents the methods performance on the 9 benchmarked domains. As noted before, the average number of modifications and runtime is comparable across methods only when the set of successful problem is the same. Refer to Figure 2 in the main paper for a problem by problem comparison.

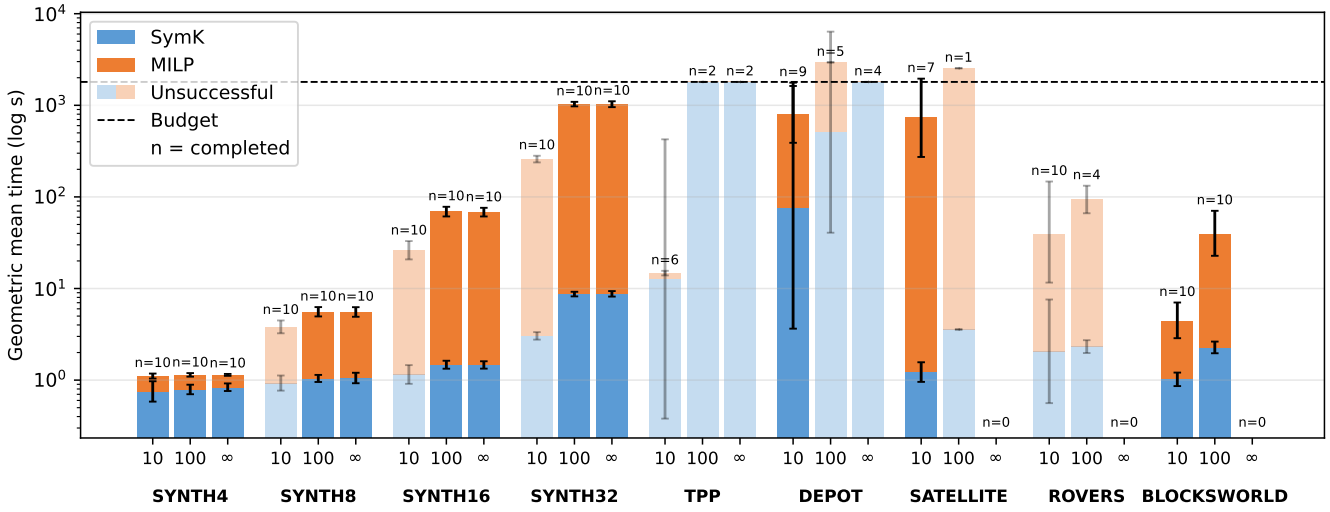


Figure 3: Execution time split into the time to compute the set of plans with SYMK (blue) and the time to compute the minimum number of modifications with the MILP (orange).

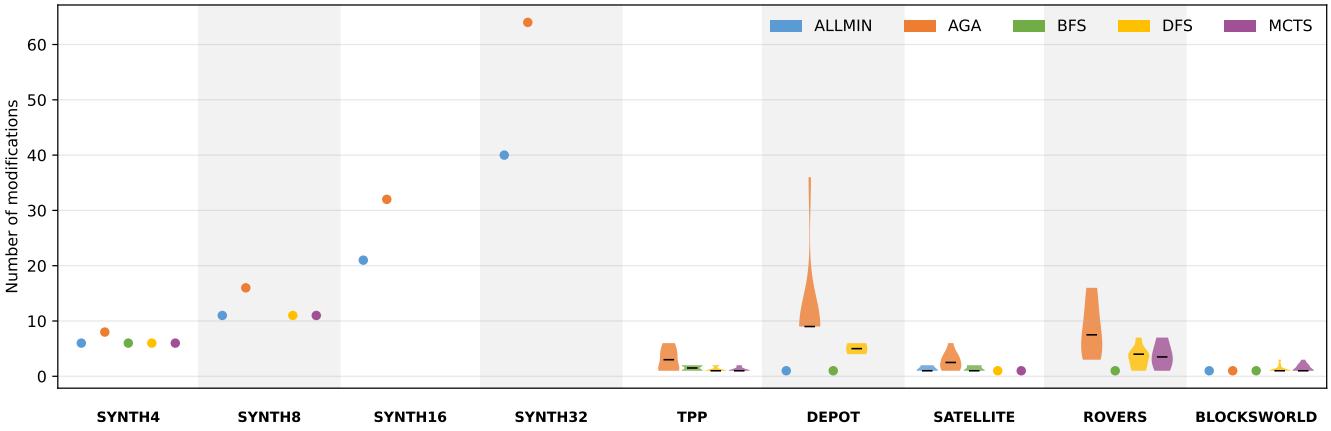


Figure 4: Number of modifications per benchmark and method. While AGA successfully renders unsolvable all benchmarks, the average number of modifications is higher.

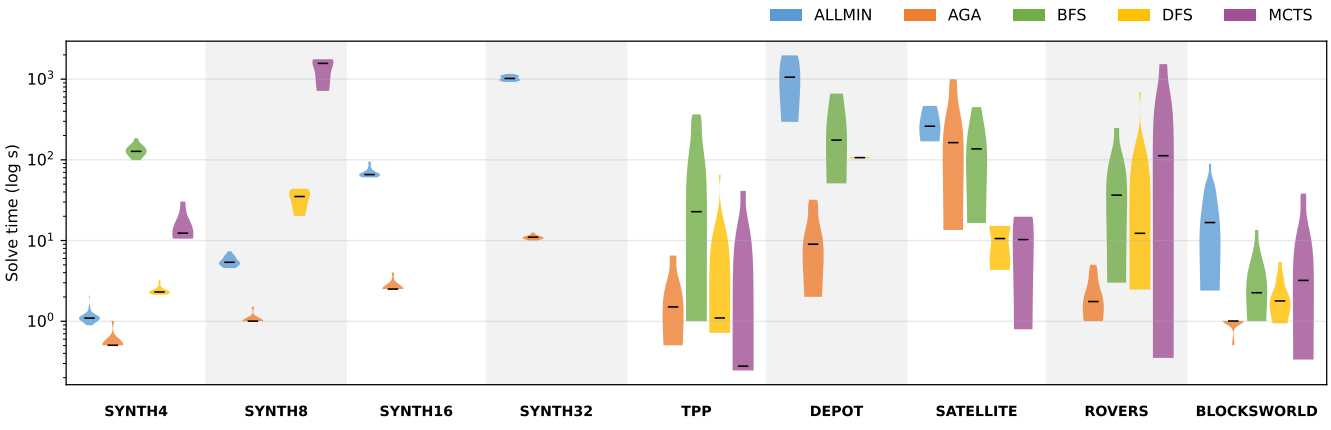


Figure 5: Mean execution time per benchmark and method. AGA is consistently faster, while MCTS shows the higher variance across the search-based methods.

Config	SYNTH4					SYNTH8					SYNTH16				
	T	C	S	$\|M\|$	Time	T	C	S	$\|M\|$	Time	T	C	S	$\|M\|$	Time
ALLMIN^{10}	10	10	**10**	$\mathbf{6.0 \pm 0.0}$	1.1 ± 0.3	10	10	0	-	-	10	10	0	-	-
ALLMIN^{100}	10	10	**10**	$\mathbf{6.0 \pm 0.0}$	1.1 ± 0.1	10	10	**10**	$\mathbf{11.0 \pm 0.0}$	5.6 ± 0.7	10	10	**10**	$\mathbf{21.0 \pm 0.0}$	69.6 ± 10.1
ALLMIN	10	10	**10**	$\mathbf{6.0 \pm 0.0}$	1.1 ± 0.1	10	10	**10**	$\mathbf{11.0 \pm 0.0}$	5.6 ± 0.8	10	10	**10**	$\mathbf{21.0 \pm 0.0}$	68.6 ± 8.7
AGA	10	10	**10**	8.0 ± 0.0	$\mathbf{0.6 \pm 0.2}$	10	10	**10**	16.0 ± 0.0	$\mathbf{1.1 \pm 0.2}$	10	10	**10**	32.0 ± 0.0	$\mathbf{2.7 \pm 0.5}$
BFS	10	10	**10**	$\mathbf{6.0 \pm 0.0}$	134.2 ± 23.7	10	0	0	-	-	10	0	0	-	-
DFS^{5}	10	0	0	-	-	10	0	0	-	-	10	0	0	-	-
DFS^{10}	10	10	**10**	$\mathbf{6.0 \pm 0.0}$	2.4 ± 0.3	10	0	0	-	-	10	0	0	-	-
DFS^{15}	10	10	**10**	$\mathbf{6.0 \pm 0.0}$	2.4 ± 0.2	10	10	**10**	$\mathbf{11.0 \pm 0.0}$	33.4 ± 8.3	10	0	0	-	-
MCTS	10	7	7	$\mathbf{6.0 \pm 0.0}$	15.8 ± 7.3	10	5	5	$\mathbf{11.0 \pm 0.0}$	1340.4 ± 461.5	10	0	0	-	-

Config	SYNTH32					TPP					DEPOT				
	T	C	S	$\|M\|$	Time	T	C	S	$\|M\|$	Time	T	C	S	$\|M\|$	Time
ALLMIN^{10}	10	10	0	-	-	10	6	0	-	-	10	9	3	$\mathbf{1.0 \pm 0.0}$	1107.4 ± 838.2
ALLMIN^{100}	10	10	**10**	$\mathbf{40.0 \pm 0.0}$	1032.4 ± 62.2	10	2	0	-	-	10	5	0	-	-
ALLMIN	10	10	**10**	$\mathbf{40.0 \pm 0.0}$	1030.8 ± 81.2	10	2	0	-	-	10	4	0	-	-
AGA	10	10	**10**	64.0 ± 0.0	$\mathbf{11.0 \pm 0.7}$	10	10	**10**	3.0 ± 2.1	2.4 ± 2.2	10	10	**10**	11.7 ± 8.5	$\mathbf{13.0 \pm 10.8}$
BFS	10	0	0	-	-	10	8	8	1.5 ± 0.5	112.8 ± 149.4	10	3	3	$\mathbf{1.0 \pm 0.0}$	296.3 ± 322.8
DFS^{5}	10	0	0	-	-	10	6	6	1.3 ± 0.5	13.7 ± 25.9	10	1	1	4.0 ± 0.0	108.5 ± 0.0
DFS^{10}	10	0	0	-	-	10	5	5	$\mathbf{1.2 \pm 0.4}$	3.3 ± 5.0	10	0	0	-	-
DFS^{15}	10	0	0	-	-	10	5	5	$\mathbf{1.2 \pm 0.4}$	4.3 ± 7.3	10	1	1	6.0 ± 0.0	104.7 ± 0.0
MCTS	10	0	0	-	-	10	5	5	$\mathbf{1.2 \pm 0.4}$	8.5 ± 18.3	10	0	0	-	-

Config	SATELLITE					ROVERS					BLOCKSWORLD				
	T	C	S	$\|M\|$	Time	T	C	S	$\|M\|$	Time	T	C	S	$\|M\|$	Time
ALLMIN^{10}	10	7	3	1.3 ± 0.6	299.1 ± 152.1	10	8	0	-	-	10	10	8	$\mathbf{1.0 \pm 0.0}$	4.4 ± 2.0
ALLMIN^{100}	10	1	0	-	-	10	4	0	-	-	10	10	9	$\mathbf{1.0 \pm 0.0}$	40.2 ± 21.8
ALLMIN	10	0	0	-	-	10	0	0	-	-	10	0	0	-	-
AGA	10	10	**10**	2.7 ± 1.6	317.6 ± 361.2	10	10	**10**	8.5 ± 4.9	$\mathbf{2.3 \pm 1.4}$	10	10	**10**	$\mathbf{1.0 \pm 0.0}$	$\mathbf{1.0 \pm 0.2}$
BFS	10	5	5	1.4 ± 0.5	178.0 ± 178.7	10	6	6	$\mathbf{1.0 \pm 0.0}$	73.6 ± 91.0	10	10	**10**	$\mathbf{1.0 \pm 0.0}$	4.1 ± 4.0
DFS^{5}	10	2	2	$\mathbf{1.0 \pm 0.0}$	9.9 ± 7.3	10	5	5	3.0 ± 1.4	145.8 ± 300.4	10	10	**10**	1.3 ± 0.7	2.3 ± 1.4
DFS^{10}	10	2	2	$\mathbf{1.0 \pm 0.0}$	$\mathbf{9.7 \pm 7.5}$	10	5	5	3.6 ± 2.3	28.1 ± 37.0	10	10	**10**	1.3 ± 0.7	2.3 ± 1.3
DFS^{15}	10	2	2	$\mathbf{1.0 \pm 0.0}$	10.7 ± 6.4	10	5	5	3.6 ± 2.3	30.0 ± 41.8	10	10	**10**	1.3 ± 0.7	2.3 ± 1.3
MCTS	10	2	2	$\mathbf{1.0 \pm 0.0}$	10.3 ± 13.4	10	4	4	3.8 ± 2.5	439.8 ± 732.1	10	10	**10**	1.5 ± 0.7	9.8 ± 13.9

Table 1: Outcome breakdown and modification counts per method and benchmark. T= total problems; C=completed, S=success. T-C=out of memory or timeout, C-S=modifications do not render the task unsolvable.