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

Alberto Pozanco, Marianela Morales, Pietro Totis, Daniel Borrajo

J.P. Morgan AI Research

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

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 ALLMIN, an optimal algorithm that solves these tasks by minimally modifying the original actions to render the planning task unsolvable. We empirically evaluate the performance of ALLMIN in shielding planning tasks of increasing size, showing how it can effectively shield the system by turning 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.

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 (and (escalated)))
```

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 $\langle$documents_submitted, application_complete, granted_approval, escalated, client_concerns, safe_client$\rangle$. 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.

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 modify 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 ALLMIN, an algorithm that computes the minimal set of modifications to the original set of actions that render the planning task unsolvable. ALLMIN computes all loopless plans that solve the planning task and then solves an optimization problem to identify the minimal set of modifications needed to invalidate all these plans.

The remainder of the paper is organized as follows. We first formalize classical planning. We then provide a formal definition of planning task shielding problems and their solutions, followed by a presentation of ALLMIN. Next, we present preliminary results for ALLMIN on a synthetic benchmark. Finally, we conclude by discussing the main results and outlining potential directions for future research.

Background

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. Its name $\text{NAME}(a)$, which is a string. A set of preconditions $\text{PRE}(a)$, which are set of fluents that need to be true for the action to be applied. Add and delete effects $\text{ADD}(a)$ and $\text{DEL}(a)$, which are set of fluents that are added (or deleted) once the action is applied. We assume $\text{ADD}(a) \cap \text{DEL}(a) = \emptyset$. And finally a cost $c(a) \in \mathbb{R}$ associated with performing the action. An action a is applicable in a state s iff $\text{PRE}(a) \subseteq s$. We define the result of applying an action in a state as $\gamma(s, a) = (s \setminus \text{DEL}(a)) \cup \text{ADD}(a)$. 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 state s is reachable from state s' iff there exists an applicable action sequence π such that $s \subseteq \Gamma(s', \pi)$. A sequence of actions is simple if it does not traverse the same state $s \in \mathcal{S}_R$ 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 $\mathcal{G} \subseteq \Gamma(\mathcal{I}, \pi)$.

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. To this end, we first formalize what it means to modify an action, and then build our notion of shielding on top of it.

Definition 2 (Action Modification)**.** *Let $a = \langle \text{NAME}(a), \text{PRE}(a), \text{ADD}(a), \text{DEL}(a) \rangle$ be an action. An action modification is a single atomic change to one of the components of a, namely the addition or removal of a fluent from $\text{PRE}(a)$, $\text{ADD}(a)$, or $\text{DEL}(a)$, producing a new action a'. Given two sets of actions $\mathcal{A}$ and $\mathcal{A}'$, we say that $\mathcal{A}'$ is a modification of $\mathcal{A}$ iff there exists a mapping between actions in $\mathcal{A}$ and actions in $\mathcal{A}'$ such that each $a' \in \mathcal{A}'$ is obtained from its corresponding $a \in \mathcal{A}$ by applying one or more action modifications. We denote by $\#(\mathcal{A}')$ the total number of action modifications applied across all actions in $\mathcal{A}$ to obtain $\mathcal{A}'$, that is,*

$$\#(\mathcal{A}') = \sum_{a \in \mathcal{A}} \big(|\text{PRE}(a') \triangle \text{PRE}(a)| + |\text{ADD}(a') \triangle \text{ADD}(a)| + |\text{DEL}(a') \triangle \text{DEL}(a)| \big)$$

where $\triangle$ denotes the symmetric difference between two sets.

With this notion in place, we define a shielding planning task and its solution as follows:

Definition 3 (Shielding Solution)**.** *Given a planning task $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, a solution to a shielding planning task $\mathcal{P}_S = \langle \mathcal{F}, \mathcal{A}', \mathcal{I}, \mathcal{G} \rangle$ is a new set of actions $\mathcal{A}'$ such that $\mathcal{P}_S$ is unsolvable.*

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.*

While trivial modifications to the original set of actions are possible, such as setting $\mathcal{A}' = \emptyset$, our goal is to identify the minimal set of action modifications that render the planning task unsolvable. We then formally define the optimality of a shielding solution as follows:

Definition 4 (Shielding Solution Optimality)**.** *A Shielding Solution $\mathcal{A}'$ is optimal iff there does not exist another solution $\mathcal{A}''$ with $\#(\mathcal{A}'') < \#(\mathcal{A}')$.*

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 actions 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,* $\text{PRE}(a) \subseteq \text{PRE}(a')$
- *removing add effects,* $\text{ADD}(a') \subseteq \text{ADD}(a)$
- *adding delete effects,* $\text{DEL}(a) \subseteq \text{DEL}(a')$

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, $\text{PRE}(a') \subseteq s$. Since $\text{PRE}(a) \subseteq \text{PRE}(a')$, it follows that $\text{PRE}(a) \subseteq s$, 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., $\text{PRE}(a) \subseteq s$), but if there exists $p \in \text{PRE}(a') \setminus \text{PRE}(a)$ 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 $s' = (s \setminus \text{DEL}(a')) \cup \text{ADD}(a')$. Since $\text{ADD}(a') \subseteq \text{ADD}(a)$ and $\text{DEL}(a) \subseteq \text{DEL}(a')$, it follows that $s \setminus \text{DEL}(a') \subseteq s \setminus \text{DEL}(a)$ and $(s \setminus \text{DEL}(a')) \cup \text{ADD}(a') \subseteq (s \setminus \text{DEL}(a)) \cup \text{ADD}(a)$. 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 $|\Pi(\mathcal{P}')| \leq |\Pi(\mathcal{P})|$. □

Apart from limiting the set of modifications, we can also limit the subset of actions in that we need to modify in order to turn $\mathcal{P}$ unsolvable. We denote by $\mathcal{A}^\Pi = \{a \mid a \in \pi, \forall \pi \in \Pi(\mathcal{P})\}$ the set of actions that appear in the plans solving the original task.

Remark 2. *Note that we only need to consider modifying actions in* $\mathcal{A}^\Pi$ *(and not the rest of the actions in* $\mathcal{A}$*) to render* $\mathcal{P}$ *unsolvable.*

Next, we present ALLMIN, our approach to compute optimal solutions to shielding 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 2. We add a fake action a^g to $\mathcal{A}$, which represents the achievement of the goals $\mathcal{G}$. $\text{PRE}(a^g) = \mathcal{G}$ and $\text{ADD} = \text{DEL} = \emptyset$. We append this action to each plan $\pi \in \Pi(\mathcal{P})$. We have the following set of decision variables:

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

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

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

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

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

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 $f \in \text{PRE}(a)$:

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

   If $f \notin \text{PRE}(a)$:

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

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

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

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 $f \in \text{ADD}(a)$:

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

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

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

   Else If $f \in \text{DEL}(a)$:

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

   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 formulation systematically 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 $\mathcal{P}$.

Evaluation

Experimental Setting

Benchmark. We use two benchmarks to evaluate ALLMIN. In the first one, we use a synthetic benchmark (SYNTHETIC) where we generate planning tasks in the form of a graph where we control: the number of plans $(8, 16, 32)$, their maximum $(4, 8, 16)$ and minimum $(2, 4, 6)$ 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 32 plans. We generate 10 random problems for each of the 3 combinations, giving us a total of 30 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, and SATELLITE. By selecting this concise yet varied set of standard planning tasks, our goal is to demonstrate how ALLMIN can be applied to transform unsolvable problems from domains that are well-known within the community.

Approaches and Metrics. We evaluate ALLMIN 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 will also evaluate two variants of ALLMIN: ALLMIN^{10} and ALLMIN^{100}. Instead of computing all simple plans, these variants compute 10 and 100 plans that solve the task, respectively. We then solve the resulting MILPs using the HiGHS solver (Hall 2019) to determine the minimal modifications required to make the planning task unsolvable. For each algorithm, we report:

1. #Solved: the number of solved instances within the time and memory bounds
2. Time (s): the total execution time (in seconds) required to generate a solution for the shielding task
3. $\#(\mathcal{A}')$: the number of modifications in the solution
4. Success: whether the returned modifications actually turn the planning task unsolvable (1) or not (0).

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

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

Table 1 presents a summary of our evaluation results. As shown, ALLMIN, which requires computing all plans that solve the original planning task, successfully solves all tasks in the synthetic benchmark but fails to solve any tasks in the standard planning benchmarks. This limitation arises because these tasks typically contain thousands of plans, and SYMK cannot compute them within the given time and memory bounds.

When we reduce the number of plans that SYMK computes, both ALLMIN^{10} and ALLMIN^{100} are able to solve more tasks, though this comes at the cost of reduced success—meaning they return solutions that may leave some plans in the original task still valid. This behavior is evident in the SYNTHETIC benchmarks. While ALLMIN^{10} computes valid shielding solutions for all tasks in the set with 8 plans (SYNTHETIC8), it does not succeed in any tasks in SYNTHETIC16 or SYNTHETIC32. The reason is that the MILP returns the minimum number of modifications for the $k = 10$ computed plans, invalidating those plans but potentially leaving other valid solutions unaffected. Although there is no guarantee regarding the correctness of the returned solution, we observe that computing a subset of all plans often enables ALLMIN^{10} and ALLMIN^{100} to achieve strong empirical performance. These methods are able to find successful modifications within a few seconds for many of the planning tasks.

These findings are summarized in Figure 1, which plots the number of problems successfully rendered unsolvable (Success $= 1$) with respect to the compute time. ALLMIN^{10} ($k = 10$) succeeds with more instances at lower compute times, that is, on small benchmarks ALLMIN^{10} is faster and considers enough plans to render the task unsolvable. The trend for ALLMIN^{100} ($k = 100$) and ALLMIN ($k = \infty$) overlaps on smaller benchmarks, i.e. the problems have less than 100 possible plans. ALLMIN^{100} succeeds in more instances than the other two variants: ALLMIN^{10} on many benchmarks lacks sufficient information about the existing plans to render the task unsolvable, while ALLMIN reaches the time limit on more instances trying to compute all plans. ALLMIN^{100} thus balances the amount of information collected about the plans and the time spent to compute it.

With respect to the number of modifications, we observe two main groups. For the standard planning benchmarks, ALLMIN and its variants typically identify a single modification that renders the planning task unsolvable ($\#(\mathcal{A}') = 1$). 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 SYNTHETIC benchmarks are more challenging, as they are designed with several independent paths leading to the goal and fewer bottlenecks. In these cases, the number of modifications is still lower than the number of plans that solve the task, indicating that ALLMIN effectively identifies actions shared across multiple plans and modifies them so that several plans become invalid simultaneously.

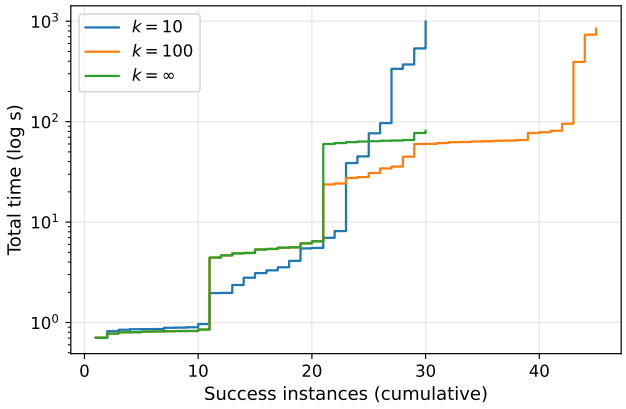

Figure 1: ALLMIN^{100} succeeds on more instances, with a better balance between available information over the plans and the time dedicated to plan computation.

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 hundred seconds for larger ones. We also 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. Figure 2 presents the results of this analysis, showing the contribution of each component to the average total execution time as the planning task size and the number of plans increase. The cost of solving the MILP dominates the time to compute the set of plans. ALLMIN terminates within the time limits on the synthetic benchmarks, although, as noted before, for SYNTHETIC16 and SYNTHETIC32 the modifications do not succeed at rendering the problem unsolvable. In none of the optimization benchmarks ALLMIN terminates within the time budget, with the SATELLITE domain being the hardest, with only one problem succeeding within the time limit at $k = 100$. This suggests that, for the current benchmark, 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.

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 modifi-

Domain (#Problems)	ALLMIN^{10}				ALLMIN^{100}				ALLMIN			
	#Solved	Time (s)	$\#(\mathcal{A}')$	Success	#Solved	Time (s)	$\#(\mathcal{A}')$	Success	#Solved	Time (s)	$\#(\mathcal{A}')$	Success
BLOCKSWORLD (10)	10	3.6 ± 1.6	1.0 ± 0.0	10	10	38.8 ± 17.7	1.0 ± 0.0	10	0	-	-	-
ROVERS (10)	8	22.4 ± 21.2	1.0 ± 0.0	3	9	304.4 ± 266.1	1.0 ± 0.0	4	0	-	-	-
SATELLITE (10)	8	57.6 ± 26.8	1.0 ± 0.0	2	2	1479.3 ± 894.3	1.0 ± 0.0	1	0	-	-	-
SYNTHETIC8 (10)	10	0.8 ± 0.1	6.0 ± 0.1	10	10	0.8 ± 0.0	6.0 ± 0.1	10	10	0.8 ± 0.0	6.0 ± 0.1	10
SYNTHETIC16(10)	10	3.2 ± 0.4	7.8 ± 0.9	0	10	5.3 ± 0.6	11.2 ± 1.5	10	10	5.3 ± 0.6	11.2 ± 1.5	10
SYNTHETIC32 (10)	10	14.8 ± 2.2	8.5 ± 0.9	0	10	66.4 ± 6.9	21.4 ± 2.1	10	10	66.4 ± 6.9	21.4 ± 2.1	10

Table 1: Average and standard deviation of the number of modifications $\#(\mathcal{A}')$ and execution Time (s) as we increase the planning task size and the number of plans that reach the flawed state $|\Pi(\mathcal{P})|$. Time is averaged over solved tasks.

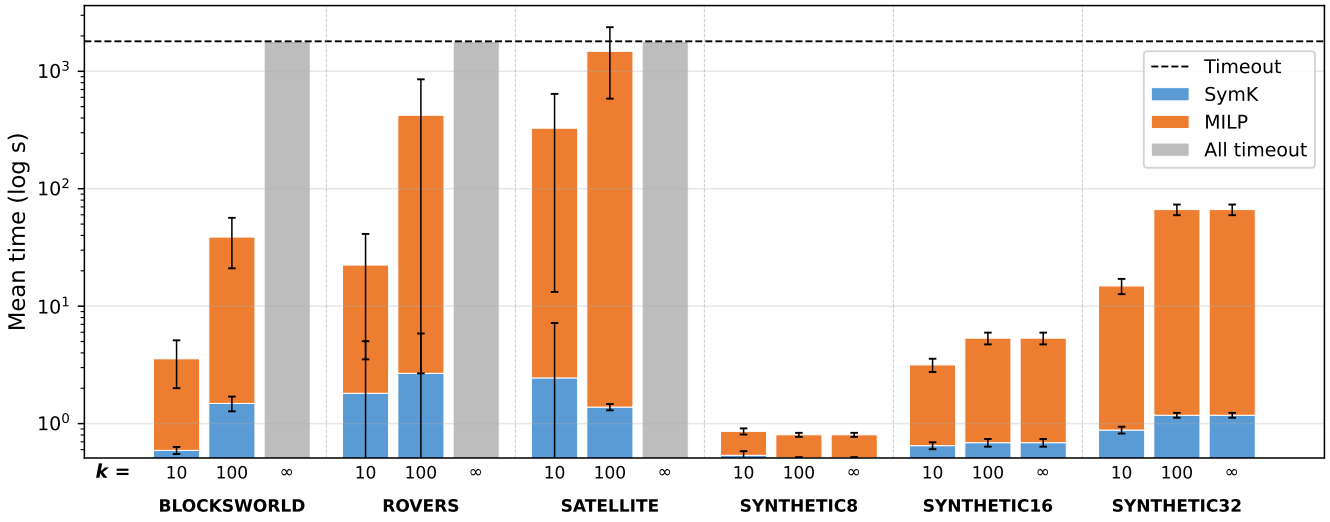

Figure 2: Execution time split into the time to compute the set of plans with SYMK (blue) and the time to compute the minimum number of modifications with the MILP (orange).

cations to the original actions: adding preconditions, removing add effects, and adding delete effects. We then present ALLMIN, an algorithm for solving shielding tasks that (1) computes a set of plans that solve the original planning task, and (2) determines the minimal set of modifications to the original actions needed to make the task unsolvable. Our preliminary evaluation demonstrates that ALLMIN can effectively render planning tasks unsolvable, thereby shielding the system and preventing the existence of plans that reach flawed states.

As next steps, we want to explore incorporating constraints or preferences for certain types of modifications, as well as considering additional objectives, such as minimizing the number of actions to which modifications are applied, or minimizing the number of fluents used in the modifications. Finally, we intend to develop alternative algorithms for solving shielding tasks that can trade some theoretical guarantees for improved empirical performance.

Disclaimer

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.

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.

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.

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.

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).

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.