Mutation-guided Metamorphic Testing of Optimality in AI Planning

Quentin Mazouni, Arnaud Gotlieb, Helge Spieker, Mathieu Acher, Benoit Combemale

Mutation-guided Metamorphic Testing of Optimality in AI Planning

Quentin Mazouni[a], Arnaud Gotlieb[a,*], Helge Spieker[a], Mathieu Acher[b], Benoit Combemale[b]

[a] *Simula Research Laboratory, Oslo, Norway*
[b] *INRIA-IRISA-INSA, Rennes, France*

Abstract

Autonomous systems such as space- or underwater-exploration robots or elderly people assistance robots often include an Artificial Intelligence (AI) planner as a component. Starting from the initial state of a system, an AI planner automatically generates sequential plans to reach final states that satisfy user-specified goals. Generating plans having a minimum number of intermediate steps or taking the least time to execute is usually strongly desired, as these plans exhibit minimal costs. Unfortunately, testing if an AI planner generates optimal plans is almost impossible because the expected cost of these plans is usually unknown. Based on mutation adequacy test suite selection, this article proposes a novel metamorphic testing framework for detecting the lack of optimality in AI planners. The general idea is to perform a systematic but non-exhaustive state space exploration from the initial state and to select mutant-adequate states to instantiate new planning tasks as follow-up test cases. We then check a metamorphic relation between the automatically generated solutions of the AI planner for these new test cases and the cost of the initial plan. We implemented this metamorphic testing framework in a tool called MorphinPlan. Our experimental evaluation shows that MorphinPlan can detect non-optimal behaviour in both mutated AI planners and off-the-shelf, configurable planners.

*Corresponding author
Email addresses: quentin@simula.no (Quentin Mazouni), arnaud@simula.no (Arnaud Gotlieb), helge@simula.no (Helge Spieker), mathieu.acher@irisa.fr (Mathieu Acher), benoit.combemale@irisa.fr (Benoit Combemale)

It also shows that our proposed mutation adequacy test selection strategy outperforms three alternative test generation and selection strategies, including both random state selection and random walks through the state space in terms of mutation scores.

Keywords: Metamorphic Testing, AI Planning, Mutation Testing

1. Introduction

1.1. Context

In many real-life autonomous systems that involve decision-making, such as space-exploration or underwater robots, elderly people assistance robots or automated production planning and industrial control systems, AI planning can be used to find a sequence of actions which moves a system from an initial state into a final state that satisfies a specified goal [20]. The recent development of competitive AI planners has opened an arena in Europe [39] and the world[1] for embedding more and more of these tools into complex autonomous systems.

As illustrated in Figure 1, AI planners take as input the definition of a *domain* which describes the predicates and actions of the considered environment and a *problem* which describes the initial state of the system and a goal to reach. As output, these planners return a plan, i.e., a sequence of instantiated actions with the objects of the problem, which moves the system from its initial state to a final state compliant with a user-specified objective. AI planners are expected to be sound and complete, which means that they shall always return a valid plan, i.e., a plan which correctly satisfies the assigned goal of the problem with domain definition, when there exists one, and they shall return *"no solution"* when there is none. Some AI planners can in addition return plans which optimize a given cost function. Common cost functions to minimize are, for example, the number of actions in the plan, the execution time of the plan, or the number of resources required to execute the plan. The language in which AI planning

[1] International Planning Competition 2023 - `https://ipc2023.github.io/`

problems are expressed, called *Planning Domain Definition Language (PDDL)* [21], has been standardized to facilitate the comparison between AI planners and foster the reuse of existing complex domain and problem definitions.

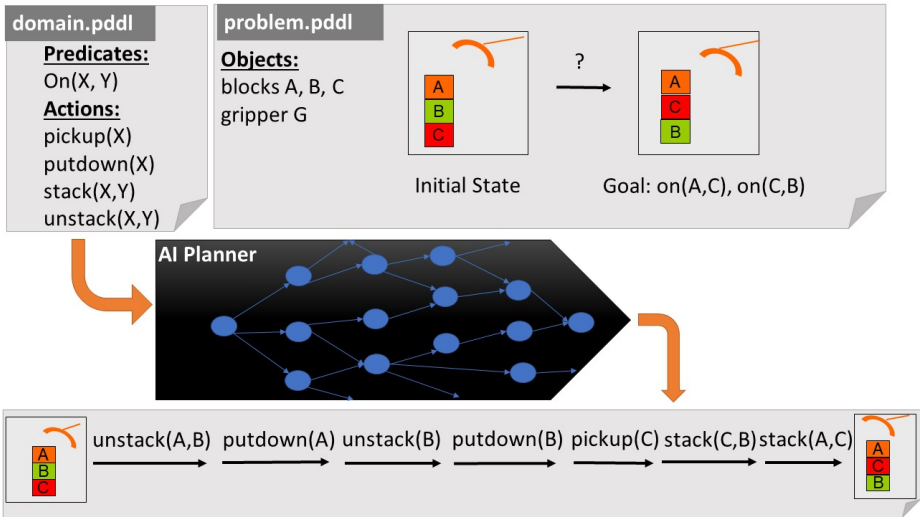


Figure 1: Solving an AI Planning problem in a nutshell.

1.2. Motivations

The development of AI planners in safety-critical missions involves strong software engineering practices such as systematic validation and verification. Many research works have addressed the problem of checking that AI planners are sound and complete [37, 6, 4, 54, 18]. However, to the best of our knowledge, the exact methods to determine if a returned plan is optimal or not do not scale up. Even though this question can be less crucial than soundness and completeness, some autonomous systems have to perform activities which span over a long-term time horizon and they cannot afford to lose time or resources by running unnecessary actions. For instance, communicating with a Mars rover takes in between 8 to 42 minutes, and it can only be done a few times during the Martian day, so the rover must compute optimal plans to save its energy and

communication resources. In production planning, computing optimal plans is often a crucial requirement to minimize the item's delivery time.

The predominant approach for verifying the results of AI planners is based on formal verification [46, 41, 4, 45, 1]. Here, the main idea is to check formal temporal logic properties of the underlying transition system by using (bounded) model checking. Although this approach is appealing, it is demanding in terms of specifications and is currently focused on verifying only safety-critical properties. Other non-functional property such as running time or memory consumption of the generated plan remains insufficiently explored in model checking of AI planners. Current methods for testing AI planners, which are less demanding than formal verification and also broadly used [27], are focused either on testing to which extent these planners deal with PDDL compliance [26, 38] or on testing their inputs domain and problem definition [40, 22, 32], i.e., ensuring that the definitions of actions do not contain contradiction which would prevent their application. Some works have also considered using AI planning to generate test cases for traditional software systems [25, 5]. But, despite the interest in verifying formally the correctness of generated plans [19] or testing domain/problem definition [22], there is a great challenge in checking whether generated plans are optimal or not. Besides detecting non-optimal plans which are unwanted, it is critical to determine if longer time or additional resources granted to a given AI planner could lead to less costly plans. This is crucial for safety- or business-critical missions with only limited time and resources to compute plans.

Testing AI planners to detect when non-optimal plans are generated is challenging as, in general, no oracle is available for a given planning problem. Apart from simple domains and problems where analytical optimal solutions can be computed (e.g., Hanoi tower problems [17]), optimal solutions are unknown and computing those would require exploring the overall state space, which is infeasible in general. An alternative can be to use differential testing [11] by launching several distinct planners and comparing the costs of the generated plans. A difference between the costs would reveal a fault in at least one of the planners.

However, this approach is not suitable for autonomous systems when tests have to be performed online, as only limited computational resources are available. AI planners are very demanding tools in terms of memory as they have to explore large state spaces, and it is often too costly to concurrently launch several planners on a single machine. Besides, AI planners for autonomous systems are usually meant to run on specific hardware and architecture, which does not facilitate the usage of multiple planners.

1.3. Contributions of the Article

In this article, we alleviate the problem by proposing mutation-guided metamorphic testing for detecting non-optimal AI planners. *Metamorphic Testing* is a testing technique introduced in [8] to replace test oracle checking with *metamorphic relations (MRs)* to assess the results of multiple program executions [43, 9]. Metamorphic Testing is used to generate so-called *follow-up test cases* [8, 10] using MRs, and it was successfully deployed to test a variety of complex software systems including ML systems [50, 51, 52, 47, 49], scientific software [53, 28], virtual reality applications [13] or autonomous vehicles [56, 14, 2]. Using mutation-guided test case selection, we generate follow-up test cases (from reachable states of the state space) from an initial problem. Then, we check the cost of the initially generated plan and the costs returned by the solutions generated by the planner under test, from the follow-up test cases. By assuming the optimality of the AI planner, our metamorphic relation states that the overall cost of the initial plan must be lower than or equal to the cost of the plan for any follow-up test case added to the cost for reaching the corresponding state. We evaluated our framework by comparing its ability to detect non-optimal behaviour in two off-the-shelf, configurable planners, considered under 20 different configurations. Our mutation-adequacy strategy led to better detection of non-optimal plans and significantly lowered the number of required follow-up test cases, as compared with simpler baseline strategies including random generation. However, its running time is also significantly higher than these simpler strategies and thus it is crucial to control its usage as discussed in the article.

The key contributions are:

- We present a mutation-guided metamorphic testing framework to detect non-optimal planning. It generates follow-up test cases from a set of states selected after an exploration of the planning problem's state space. Violations of the metamorphic relation let us detect non-optimal planning and thus reveal faults in the program implementation.

- We handle the follow-up test case generation in two steps: state space exploration and state selection. The selection step scores the explored states with their ability to kill mutated AI planners. We choose these mutants by combining mutated functions leveraged in a classical A^* search. This approach (distinct from traditional mutation operators) lets us preserve the validity of the mutated programs' outputs.

- We implemented this framework in a tool called MorphinPlan. To the best of our knowledge, it is the first tool of that kind, able to detect non-optimal plans returned by existing AI planners.

- Using mutation testing, we conduct a two-stage empirical evaluation of our test case generation method, for testing two off-the-shelf planning systems, including Fast Downward. We show that MorphinPlan can successfully generate test cases that detect non-optimal plans in mutated AI planners and existing configurations. The evaluation compares the mutation adequacy state selection strategy with three alternative state selection strategies used as comparison baselines and reveals an improved rate of fault-revealing follow-up test cases, thus improving the efficiency of testing.

1.4. Plan of the Article

The article is organized as follows: Section 2 introduces AI planning and Metamorphic Testing with an illustrative example. Section 3 introduces the Metamorphic Testing framework. Section 4 describes the MorphinPlan tool

while Section 5 presents our empirical evaluation. Threats to the latter are treated in Section 6. Section 7 compares our work with other approaches. Eventually, Section 8 concludes the article and draws some perspectives.

2. Background

2.1. AI Planning

Given an initial state and a goal to reach, AI planning aims at generating a plan from the initial state to a final state which satisfies the goal. A *plan* (Def.2) can be seen as a sequence of successive actions in a *state transition system* (Def.1).

Definition 1 (State Transition System)**.** *A State Transition System is a tuple* $\Sigma = (S, A, c, \gamma)$ *where* S *is a set of states,* A *is a set of actions,* $c : A \longrightarrow \mathbb{R}^+$ *is a cost function,* $\gamma : S \times A \longrightarrow S$ *is a state transition function.*

In a given state s, only a limited number of actions can usually be applied and thus γ is only a partial function. When $\gamma(s, a)$ is defined, we say that the action a is *applicable* in the state s and that it results in a new state s' with cost $c(a)$.

Definition 2 (Plan)**.** *A plan is a sequence of actions* $\pi = [a_1, .., a_k]$ *s.t. each action of* π *is applicable from an initial state* s_0 *of the system. Applying* π *to* s_0 *leads to the following sequence of states:* $s_1 = \gamma(s_0, a_1), s_2 = \gamma(s_1, a_2), \ldots, s_k = \gamma(s_{k-1}, a_k)$*. In that context,* s_k *is called the final state of the system after the application of* π*. Note that the size of* π *is* k*, noted* $|\pi|$ *and its cost is* $\sum_{i=1}^{k} c(a_i)$*, noted* $c(\pi)$*.*

A state s is reachable from an initial state s_0 if there exists a plan $\pi = [a_1, .., a_k]$ s.t. $s = \gamma(..(\gamma(\gamma(s_0, a_1), a_2)..), a_k)$. By extension, $\gamma(s_0, \pi)$ denotes the state that results from applying π to s_0. In AI planning, there are several representations of states and actions. We adopt the so-called "logical representation" where each state and action modification is captured by logical propositions. All

examples of the article use PDDL syntax [21]. We assume some basic knowledge of PDDL, even though it is not mandatory to understand the article.

A state s is represented with a conjunction of literals over constant symbols (e.g. `r`). Variables can be used to represent multiple symbols. Variables are noted with a question mark (e.g., `?r`).

Example 1. *(State)*
`(at r1 w) (loaded r1 i)` *describes a state where a known rover* `r1` *stands at a waypoint* `w` *and holds an item* `i`*, while* `(at ?r w)` *is true iff there is a rover (unknown) standing at waypoint* `w`*.*

An action a is a triple $(head(a), pre(a), eff(a))$ where (i) $head(a)$ is composed of a name and a list of variable parameters; (ii) $pre(a)$ is a set of literals which must stand for the application of the action; (iii) $eff(a)$ is a set of effects expressed with literals. These literals become true after the application of the action.

Example 2. *(Action)*
```
(:action move :parameters (?r ?w1 ?w2)
:precondition (at ?r ?w1)
:effect (and (at ?r ?w2) (not (at ?r ?w1))))
```
The `move` *action takes as inputs a rover* `r` *and two waypoints* `w1,w2`*. Given that* `r` *is at* `w1`*, the action moves the rover to* `w2`*, which is not anymore at* `w1`*. The use of variables makes the action generic for all rovers and waypoints.*

Using the logical representation of states and actions, an AI planner aims at solving *planning problems*, defined as follows:

Definition 3 (AI Planning Problem)**.** *Given P a finite set of propositions, A a finite set of actions, each of them being represented by the triple $(head(a), pre(a), eff(a))$ where $pre(a)$ and $eff(a)$ are subsets of P, a planning problem is defined as a tuple (P, A, s_0, obj) where, in addition to P and A, an initial state $s_0 \subseteq P$ is defined and an objective $obj \subseteq P$, composed of conjunction of literals is given.*

In PDDL, AI planning problems are materialized by two files, namely a *domain* file which contains the definition of predicates and actions and a *problem* file which contains the definition of the initial state and the objective. The following domain example shows the actions that can be performed by a very simple rover.

Example 3. *(Rover Domain Definition)*

```
(define (domain Rover)
(:predicates (rover ?r) (free ?r) (wp ?w) (item ?i) (at ?r ?w)
(loaded ?r ?i))

(:action load :parameters (?r ?w ?i)
:precondition (and (rover ?r) (wp ?w) (item ?i) (at ?r ?w) (at ?i ?w)
(free ?r))
:effect (and (loaded ?r ?i) (not (free ?r)) (not (at ?i ?w))))

(:action unload :parameters (?r ?w ?i)
:precondition (and (rover ?r) (wp ?w) (item ?i) (at ?r ?w) (loaded ?r ?i))
:effect (and (free ?r) (at ?i ?w) (not (loaded ?r ?i))))

(:action move :parameters (?r ?w1 ?w2)
:precondition (and (rover ?r) (wp ?w1) (wp ?w2) (at ?r ?w1))
:effect (and (at ?r ?w2) (not (at ?r ?w1)))))
```

Given an AI planning problem (P, A, s_0, obj) in the logical representation, the *state space* of the problem materializes as a transition system $\Sigma = (S, A, c, \gamma)$ where $S \subseteq 2^P$ is a subset of states on P. Then, solving an AI planning problem means finding a plan to go from the initial state to a state which satisfies the objective.

Definition 4 (Solution and Optimal Solution)**.** *A solution to an AI planning problem (P, A, s_0, obj) is a plan π such that $obj \subseteq \gamma(s_o, \pi)$. The cost of a solution is simply $c(\pi)$. An optimal solution of (P, A, s_0, obj) is a plan π^* such*

that $\pi^* = \arg\min c(\pi)$.

Note that there can be multiple optimal solutions. Finding solutions to an AI planning problem is known to be NP-hard [20] and most classical AI planners combine a search engine based on variations of the A* algorithm with heuristics and forward/backward strategies to explore the state space.

The following code gives an example of a typical AI planning problem where an initial state is described with one free rover `r1`, one item `i1` and two waypoints `w1,w2`, where both `r1` and `i1` are at `w1` and the objective is to find an optimal plan which leads the system to a state where the item `i1` is transported to `w2` and the rover `r1` is at `w1`.

Example 4. *(A planning problem)*

```
(define (problem rover1) (:domain rover) (:objects r i w1 w2)
(:init (rover r) (free r) (item i) (wp w1) (wp w2)
(at r w1) (at i w1))
(:goal (and (at i w2) (at r w1))))
```

Using a typical AI planner, for example, the online planner **planning.domains**[2] [34], will give the following optimal solution:

Example 5. *(An optimal solution)*

```
[(load r w1 i),(move r w1 w2),(unload r w2 i),(move r w2 w1)]
```

2.2. Metamorphic Testing (MT)

MT aims at (i) checking the expected outputs of a system under test by using user-defined properties of the system, called *metamorphic relations* and (ii) generating *follow-up test cases* by using these relations [8, 9]. In the following, we formalize MT by reusing the notations of [47].

Definition 5 (Metamorphic Relation (MR))**.** *Let f be a target function, an MR is a necessary property over a sequence of multiple inputs $\langle x_1, .., x_n \rangle$ ($n \geq 2$)*

[2]Online at: http://planning.domains/

and their corresponding outputs $\langle f(x_1), .., f(x_n)\rangle$*. MR is expressed as a relation* $\mathcal{R} \subseteq X^n \times Y^n$*.*

Definition 6 (Follow-up test cases)**.** *Let the following* $\mathcal{R}(x_1, .., x_n, f(x_1), .., f(x_n))$ *be an MR, then the sequence of inputs and their corresponding outputs defines the set of source inputs. For each source input* $\mathcal{S}(x) \in \mathcal{R}$*, a follow-up test case* $\mathcal{F}$ *is derived by applying a possibly non-deterministic transformation function* T *to the input of the source test case* x*:* $\mathcal{F}(x) = f(T(x))$*. The transformation function* T *is constructed such that the follow-up test case* $\mathcal{F}$ *fulfills the necessary property of* $\mathcal{R}$*.*

Let P be an implementation of f and $\mathcal{R}$ be the MR $\mathcal{R}(x_1, .., x_n, f(x_1), .., f(x_n))$, then applying MT for P involves the following steps:

1. Define $\mathcal{R}'$ by replacing f by P in $\mathcal{R}$.

2. Given a sequence of source test cases $\langle x_1, \ldots, x_k\rangle$, execute them to obtain their respective outputs $\langle P(x_1), \ldots, P(x_k)\rangle$. Construct and execute a sequence of follow-up test cases $\langle x_{k+1}, \ldots, x_n\rangle$ according to $\mathcal{R}'$ and obtain their respective outputs $\langle P(x_{k+1}), \ldots, P(x_n)\rangle$.

3. Examine the results with reference to $\mathcal{R}'$. If $\mathcal{R}'$ is not satisfied, then this MR has revealed a fault in the implementation of P.

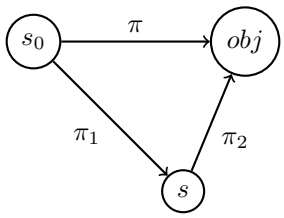
Using $\mathcal{R}$ as MR, the follow-up test case $\mathcal{F}$ with test input $T(x)$ can be generated using so-called *generators*. As indicated in Def.6, the transformation T is not necessarily a deterministic function, and it can lead to the production of multiple test cases.

3. Mutation-guided MT of AI Planners

This section presents an automated mutation-guided MT method dedicated to the detection of non-optimal AI planners.

3.1. An MR for Detecting Non-Optimality

We base our MR on a well-known property of optimal plan which establishes that, for a given supposedly optimal initial plan, there cannot be strictly less costly plans obtained by transiting by another reachable state than the initial one. As illustrated below, if π is the initial plan returned by the planner under test (PUT), then the cost of any plan π_1 transiting by a state s added to the cost of any plan π_2 cannot be less than the cost of π.



For a given implementation *planner*, an MR for testing non-optimality can be formally defined as follows:

Definition 7 (MR over *planner* w.r.t. s)**.** *Let $p = (P, A, s_0, obj)$ be an AI planning problem, s be a reachable state ($\neq s_0$) and planner be a supposedly optimal AI planner, then an MR for p w.r.t. s is*

$$\langle p, p' \rangle \implies c(planner(p)) - c(planner(p') \leq c([a_1, .., a_k])$$

where $[a_1, ..a_k]$ is a plan from s_0 to s, p' is defined as the AI planning problem $p' = (P, A, s, obj)$ and $planner(p)$ represents the call of planner on the problem p. If we consider only unitary cost (i.e., the cost of each action equals 1.) then the formula simplifies as

$$\langle p, p' \rangle \implies (|\pi| - |\pi'|) \leq k$$

where k is the number of actions performed to reach s from s_0, π (resp. π') is a plan generated by planner from s_0 (resp. s).

Note that any reachable state (but the initial state of the problem) is eligible to reveal the non-optimality of the planner. Other MRs can be used for testing

AI planners, such as, for instance, relations that change the order of defined actions in the domain definition file or the order of objects in the problem definition file, but these relations would be useless to test for optimality as most AI planners just preprocess their inputs to put actions and objects in a predefined similar order. The MR from Def. 7 captures the essence of optimality issues in AI planning. Two main approaches can be used to explore the state space: progressing in forward chaining from the initial state (it corresponds to reaching s and asking the PUT to produce π_2), or regressing in backward chaining from the objective (thus reaching s from obj and asking the PUT to produce π_1). Note that we can also choose s arbitrarily and let the PUT generate both π_1 and π_2 but this approach would be highly inefficient as the likelihood to choose a reachable state is very low. In our work, we use only the former approach, as it explores reachable states only. Indeed, the backward approach that was also implemented and evaluated leads to inefficiently exploring too many unreachable states. We then gave up on this backward exploration approach. Note that the absence of violation of this MR does not ensure that the AI planner under test is always optimal as 1) the definition is related to selected planning tasks which means that other tasks may reveal undetected optimality faults; 2) checking the MR for all possible states s is intractable as exploring the whole state space is usually impossible and only a selection of states can be tested. Furthermore, another limitation of this approach lies in the inconsistency of the subsequent follow-up planning tasks. For some states s, the plan π_2 does not necessarily exist and then checking the MR for s is useless.

3.2. Mutation Adequacy Test Case Selection

Generating follow-up test cases with the proposed MR (Def. 7) involves 1) generating states by exploring the state space from the initial state (using the forward chaining approach); 2) selecting the states which are the most error-prone; and 3) launching of new planning tasks from these states. By observing that there is no obvious relation between states and optimality fault detection, we then face a *search problem* without benefiting from any guidance on how to

Algorithm 1 Mutation adequacy scoring computation

Input: MUT: a set of mutated AI planners, Task: an AI planning problem

Output: Scores: List of scores

1: $Costs \leftarrow [\,]$
2: **for all** $m \in MUT$ **do** ▷ Collect source costs
3: $\pi \leftarrow execute(m, Task)$;
4: $Costs.append(c(\pi))$
5: **end for**
6: $Nodes \leftarrow generateStates(Task)$
7: **for all** $(s, c) \in Nodes$ **do** ▷ Mutation score per state
8: $score \leftarrow 0$;
9: $Subtask \leftarrow PlanningTask(Task, s)$
10: **for all** $m \in MUT$ **do**
11: $\pi \leftarrow execute(m, Subtask)$
12: **if** $Costs[m] > c + c(\pi)$ **then** $score \leftarrow score + 1$
13: **end if**
14: **end for**
15: $Scores.append(cost)$
16: **end for**
17: **return** $Scores$

select states. Our approach to tackling this problem is to guide state selection by using mutation testing. We guide the state selection by scoring the states with their ability to kill mutated AI planners. A mutant is killed by a state if the subsequent test case violates the MR.

Algorithm 1 details our mutation adequacy scoring computation. The algorithm starts by executing all AI mutated planners on the input AI planning problem and saves the costs of their generated plans, i.e., the *source costs* (lines 1-5). Then, the state space of the problem is explored to collect potential states useful for generating follow-up test cases (line 6). Each returned state s is labelled with the cost c needed to reach it from s_0. Next, we iterate over

the collected pairs (s, c) to compute their mutation adequacy score by first instantiating the associated follow-up AI planning problem (line 7-9), second, by executing all mutated AI planners (line 10-11) on this newly generated test problem, and finally by retrieving the costs of their generated plans. Then, the algorithm checks the MR and counts the number of its violations, leading to the adequacy score (line 12).

The analysis of Alg. 1 shows that its complexity is $O(|MUT|\cdot|Exec|\cdot|Nodes|)$ where $|MUT|$ denotes the number of mutated AI planners, $|Exec|$ denotes the number of calls to an AI planner and $|Nodes|$ denotes the number of generated states. Note that all mutated AI planners are invoked once for solving each generated AI planning problem (for each newly generated state). Hence, in the proposed framework, it is necessary to limit the number of generated states ($|Nodes|$) such that the complexity of the computation remains bounded. Note also that even though the cost of the state generation (call to *generateStates* in line 6) can vary depending on the chosen exploration strategy and the considered AI planning problem, the overall time complexity of Alg. 1 is dominated by the calls to the mutated AI planners ($|Exec|$). Luckily, these calls can be artificially limited by setting up acceptable time-outs.

3.3. Workflow of the MT framework

The overall workflow of our mutation-guided MT method is shown in Figure 3.3. In addition to the AI planner under test PUT, the method takes as inputs *source input* (the initial planning problem), N_{max} (the maximum number of follow-up test cases) and a set of mutated AI planners *mutants*.

Initialization The PUT is executed on *source input* and the cost of the generated plan, *source output*, is computed.

State Generation This step explores the state space and gathers visited states. As said in Section 3.1, the process starts from the initial state and progresses through the state space by using forward chaining. By default, the exploration performs a breadth-first search, since it discovers the states with the shortest paths. The states are then saved and labelled with the cost of the plan used to

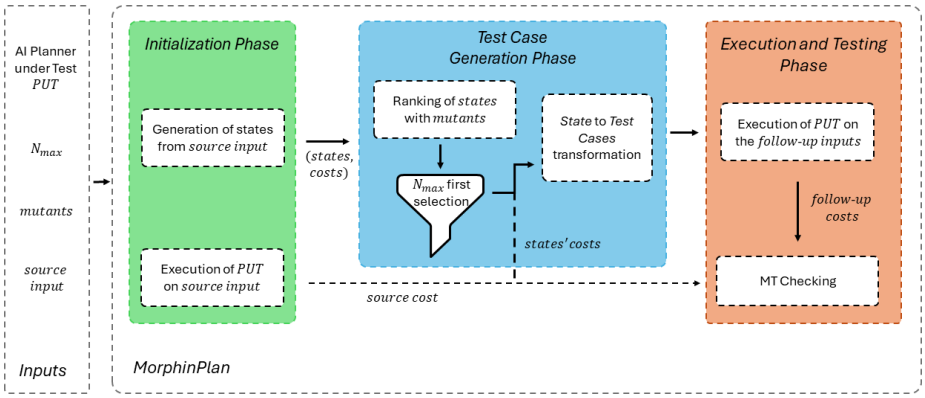


Figure 2: Overview of the MT Framework. As inputs, it receives the AI Planner under test PUT, the maximum number of follow-up test cases N_{max}, a set of mutated AI planners *mutants* and an initial planning task *source input* (expressed in PDDL). The boxes summarise the main stages that are sequentially executed: initialization and state generation (in green), state selection and their transformation to follow-up test cases (in blue), MT testing of PUT (in orange). Bold arrows denote the workflow, while dashed arrows note computed data that is reused in later phases.

reach them. Note that, as the state space can be very large for some problems, the generation must be limited. In our implementation, we stop the generation when a user-specified number of states has been reached, but other stopping criteria can be used, such as a pre-selected time-out.

State Selection The generated states are first scored and ranked *w.r.t.* their respective adequacy score on the set of mutated AI planners. Then, the method returns the N_{max} first states whose costs are lower than the *source output*'s one.

Follow-up Input Generation Each selected state is used to create a new planning problem, i.e., a follow-up test case, by producing a new PDDL problem file. Given a generated state s and the *source input* (P, A, s_0, obj), the follow-up test case is the planning problem (P, A, s, obj).

Execution This step eventually executes the PUT on the set of follow-up test cases, retrieves its outputs, called *follow-up outputs*, and returns their respective costs.

Validation The MR is checked for every *follow-up output* cost, that is, its sum with the cost of the related state must be higher or equal than the cost of the

source output.

4. Implementation and Experimental Setup

4.1. Implementation

We implemented our MT framework for detecting optimality faults in AI planners in a tool called MorphinPlan. The implementation avoids redundant computations as much as possible by saving all the generated states for each planning task as well as the results of the mutated AI planners. The tool is implemented in SICStus Prolog [7] in approximately 4KLOC and it can test any PDDL2.1-based AI planner, provided that there are no durative actions and preconditions, and no continuous effects. In MorphinPlan, mutation testing is used at two distinct levels: i) to implement the mutation-based test case selection strategy and ii) to test MorphinPlan w.r.t. its capability to detect optimality faults in generated plans by AI planners. Therefore, for both cases, we created 22 non-optimal by-design AI planners, i.e., *mutated planners*, from a *reference AI planning implementation* to create source-mutated AI planners. These mutants are precisely described in Section 4.3 as designing functional mutants which exhibit optimality faults is not a trivial task.

To implement our mutation adequacy test case selection strategy, we considered a state generation based on a breadth-first search exploration with a maximum boundary of 500 states. In our experiments, we observed that varying this number by 20% has no impact on the results.

Following the Open Science Policy, we provide the source code of MorphinPlan as well as all the needed material to replicate the results presented in the paper. The MorphinPlan repository is `https://github.com/QuentinMaz/MTFramework/tree/feature/fse` and the 22 mutated planners are available in `https://github.com/QuentinMaz/MutatedAIPlanner/tree/develop`.

4.2. Experimental Setup

Our experiments consist of executing MorphinPlan on AI planners under test (PUTs) on a selection of PDDL AI planning problems. To address as best

possible the huge variety of AI planning problems, we selected a subset of PDDL problems from those used in the international planning competition (IPC).[3] The problems were selected on 1) their diversity by picking up problems coming from different application areas (i.e., domains such as block-world, robotics, logistics, etc.) and 2) the current restrictions of our reference AI planner implementation which supports only PDDL2.1 with no durative and continuous actions. The selection resulted in 19 problems from 10 distinct domains, which are listed in Table 1 with their reference.

4.3. Mutated AI Planners

To support mutation-guided MT and to perform mutation testing of MorphinPlan, we created hard-coded non-optimal-by-design AI planners from Sasak's PPL, an existing open-source Prolog-based AI Planning library [42]. Since applying standard mutation operators (e.g., Relational or Boolean Operator Replacement) to AI planner implementations had almost invariably led us to unusable planners (i.e., planners that cannot generate any plan), we manually designed 22 mutated AI planners intending to generate non-optimal plans in some cases. Note that creating such mutated AI planners is not trivial, as it requires the introduction of subtle faults that lead the planners to produce a valid but not optimal plan, i.e., a plan which satisfies the objective but has a strictly higher cost than an optimal plan. Our idea was to implement a forward A^* search that does not reopen the closed set (i.e., states that have already been visited), and where the f function (used to prioritize the open set) and the h function (used to approximate the distance of a state to the goal) are mutated. We developed 5 f and 6 h functions, which are presented in Table 2. We then compared the plans of the 30 (f, h) combinations to filter *equivalent mutants* [31]. In our context, a mutated planner is an *equivalent mutant* [31] if and only if, despite the introduced modification, the mutated planner generates only optimal plans for all the considered planning problems, as done by the

[3] `https://www.icaps-conference.org/competitions/`

Domain	Description
airport	Ground traffic control on airports [24].
blocks	Blocks world: Stackable blocks must be re-assembled on a table [3].
gripper	A robot with two grippers is tasked to take balls from one room to another [33].
elevator	Elevator control: Transport passengers with an elevator under side constraints [3].
openstacks	Minimum maximum simultaneous open stacks: Order production tasks such that there is only a minimal number of open orders [16].
pegsol	Peg Solitaire game [35].
psr-small	Power Supply Restoration: Resupply number of lines in a faulty electricity network [24].
tpp	Travelling Purchaser Problem [16].
transport	Logistics planning problem used in the IPC 2008 competition [35].
travel	Travel logistics involving side constraints.

Table 1: Overview of the planning domains used in the experimental evaluation.

original implementation. After filtering, we ended up with 22 distinct mutated planners, as shown in Table 3.

4.4. Baseline Strategies

We have identified three simple strategies, alternative to our mutation-based test case selection method, as comparison baselines. These strategies replace the state generation and/or the state selection components. For selecting N states as follow-up test cases, we consider the following strategies:

bfs_det Breadth-first search from s_0 and selection of the N first-found states;

bfs_ran Breadth-first search from s_0 and random selection of N visited states among 500 generated states;

wal_ran Iterative random walk from s_0 repeated N times. The maximum length of each walk is limited to the cost of the *source plan*;

These baselines are complementary as they correspond to three non-costly strategies for generating/selecting states in the state space. The first one systematically explores the state space from s_0 and evaluates whether further exploration is needed. The second one challenges whether all states are equally capable of revealing optimality faults. Eventually, the third baseline strategy evaluates the general relevance of a dedicated state generation and selection functionality, while a simple random walk exploration can be used.

5. Experimental Evaluation

5.1. Research Questions

The evaluation of our mutation-guided MT framework led us to answer the three following research questions (RQs):

RQ1 How effective is the proposed framework at detecting optimality faults in mutated AI planners?

RQ2 How efficient is *mutation adequacy test selection* at detecting optimality faults, as compared to identified baseline strategies?

RQ3 Can the framework detect optimality faults in off-the-shelf AI planners?

We conducted two main experiments to answer the three RQs. For the first experiment, which aims to answer RQ1 and RQ2, we randomly split the set of mutated AI planners: some of them become PUTs to evaluate the capability of MorphinPlan to reveal optimality faults in AI planners, and the others serve to guide the mutation adequacy test selection strategy. To mitigate bias on the mutated AI planners' selection for one or the other task, we report averaged

results over 20 executions with different random splits on the set of mutated planners.

Regarding the second experiment, which answers RQ3, we used MorphinPlan to test existing off-the-shelf planners and used all the mutated AI planners available to guide the test selection. Since MorphinPlan can test any PDDL2.1-based AI planner, we decided to source the PUTs from two versions of a reference AI planner, namely the PPL planner described in Section 4.3 and several distinct configurations of a well-recognized implementation, namely Fast Downward[4] [23], which won one of the IPC 2018 competition track. Fast Downward is one the most used planning systems in research, with more than a decade of contributions from the AI planning community.[5]

For both experiments, we ran MorphinPlan with $N_{max} = 4$ which means that only 4 follow-up test cases were considered for detecting non-optimal plans. In subsequent runs of MorphinPlan dedicated to analysing the importance of the N_{max} value in the process, we varied that value from 1 to 60 (see Figure 4). We systematically measured the runtime needed for the generation and selection of test cases in all cases. We excluded the time taken by the mutated AI planners as 1) it is artificially limited by a parameterized time-out value and 2) to keep the comparison fair w.r.t. the other strategies which do not call the mutated AI planner.

All our experiments were performed on a Linux machine (Ubuntu 22.04.3 LTS) equipped with an AMD Ryzen 9 3950X 16-Core processor and 32GB of RAM.

5.2. RQ1: Effective detection of mutated AI planners

To answer RQ1, we measure mutation scores and running times achieved by MorphinPlan on the aforementioned 19 AI planning problems. The mutation scores are reported as percentages of mutants detected as non-optimal by at

[4]Fast Downward: `https://www.fast-downward.org`

[5]Brief history of Fast Downward: `https://github.com/aibasel/downward/tree/main?tab=readme-ov-file#contributors`.

least one follow-up test case. Table 4 1^{st} column presents the average results over the 20 subsets of validation/selection mutants obtained with MorphinPlan, as well as the execution time. We observe that MorphinPlan gives good averaged mutation scores on most problems. More precisely, 15 over 19 problems have revealed non-optimality in the AI mutated planners, with averaged mutation scores greater than 80% with acceptable deviation (around 5% in most cases). gripper01, psr-small08 and tpp03 have mutation scores close to the threshold of 80% (around 76%), with deviations around 9%. openstacks01 has only a mutation score of circa 27%, which will be discussed in RQ2, along with the results of the baseline strategies. As the mutated AI planners used to guide MorphinPlan have been randomly selected among the 22 available mutants, our experiment shows that the mutation-guided MT framework is agnostic w.r.t. the selection of mutants. It also shows that the framework is efficient in detecting non-optimality in mutated AI planners by using any PDDL domain/problem. Only one problem (i.e., openstacks01) exhibits a low mutation score, while almost all others achieve scores greater than 80%. Regarding execution time, Table 4 shows that the execution time of MorphinPlan remains acceptable as it ranges between $20s$ and $214s$ with an average of $60s$. This mean value is an acceptable execution time for an auto-test procedure to be embedded into an autonomous system. Note that this mean value can also be considered regarding the time needed to execute the planner to let it compute an optimal plan. Even if the time needed is artificially bounded, it is usually set to longer time slots (typically $360s$ or more).

5.3. RQ2: Efficiency of mutation adequacy test selection as compared to simpler strategies

In Table 4, we report the average mutation scores achieved by the three baseline strategies, namely bfs_det, bfs_ran and wal_ran along with their execution times. These results show that MorphinPlan performs best on all of the 19 problems, with a mean of 84.7%. Looking at the mean results, we see that both bfs-based strategies achieve close mutation scores of $\sim$52% while wal_ran

achieves only ∼34%. The result obtained in openstacks01 shows that all four strategies achieve poor mutation scores. The explanation lies in the hardness of killing some mutants with this problem. As the two non-deterministic strategies (i.e., bfs_ran and wal_ran) outperform bfs_det, it seems that the mutants killed by all these three strategies are similar, and the other mutants remain alive for this problem. To refine our analysis, we also looked into the percentage of generated follow-up test cases that lead to a violation of the MR (i.e, test cases that can detect the generation of non-optimal plans). Figure 3 shows the averaged results over the 19 problems compactly. We observe that MorphinPlan achieves a much higher percentage (higher than 70%) than the three other strategies (less than 28%). It means that the ability to generate fault-revealing test cases is much higher for MorphinPlan. Regarding running times, the mean time needed by the alternative strategies is much less for bfs_det and bfs_ran than for MorphinPlan. It is understandable as MorphinPlan explores the state space up to 500 states, while the other strategies limit their exploration to N_{max}. However, as discussed in Section 5.2, the mean time taken by MorphinPlan is acceptable as it never exceeds the auto-test time limit of $360s$.

By studying the influence of the N_{max} value over the mutation scores results, as shown in Figure 4 (right plot), our experiment shows that the baselines' scores increase with the number of test cases executed. However, MorphinPlan quickly achieves a high mutation score with only a few follow-up test cases, and stabilizes its results from $N_{max} \geq 10$. This swift convergence emphasizes the overall efficiency observed in Figure 3 and reveals that the number of test cases needed to achieve high mutation scores is much less than the three competitive baselines. Regarding RQ2, not only the mutation adequacy state selection strategy leads to better non-optimal behaviour detection compared to the baselines, but also lowers the needed number of follow-up test cases, underlining the efficiency of mutation adequacy test selection for our MT framework. However, it is stressed that the time needed to execute that strategy is much higher than the one needed for simpler strategies. It is thus crucial to control

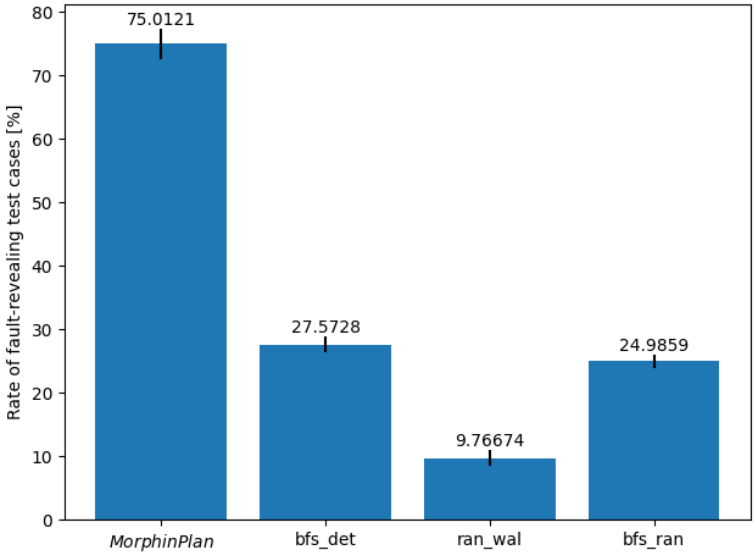


Figure 3: Percentage of MR-violating test cases for the different strategies. The results are averaged across the 19 problems considered.

it by appropriate time-out values.

5.4. RQ3: Detection of non-optimality in off-the-shelf planners

To answer RQ3, we synthesize the results obtained by our second experiment, which are reported in Table 5. Regarding off-the-shelf planners, we considered three configurations of the Sasak's PPL (V0, V1-A_{h0} and V1-A_{hmax}) and a total of 17 configuration settings for Fast Downward, i.e., using different internal parameters. Among these 17 configurations, 7 are using heuristics and algorithms that lead the planner to generate only optimal plans (column "Opt") and 10 are using non-admissible heuristics. We considered these configurations as other baselines to check whether MorphinPlan can detect these non-optimal planners. Interestingly, we considered V0 of Sasak's PPL because it contains a subtle known optimality fault that was later corrected in V1. We wanted to see if MorphinPlan could retrieve that bug. Table 5 shows the results of non-optimality detection per AI planning problem. First, note that

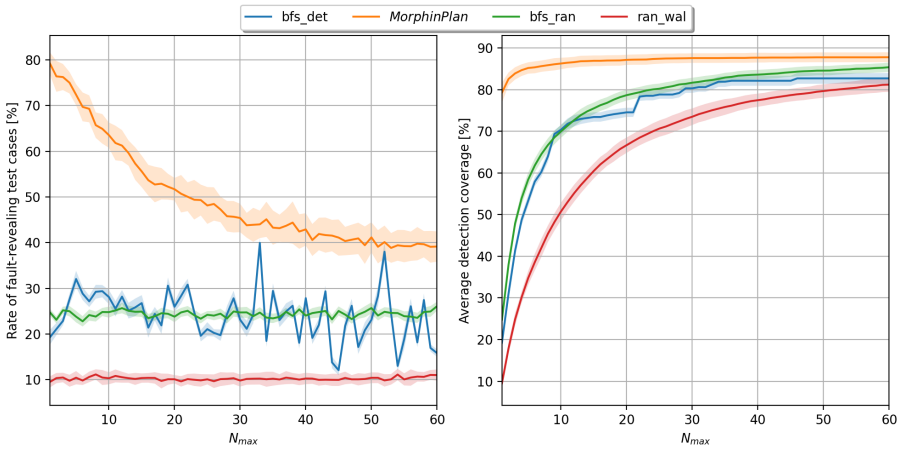


Figure 4: Evolution of the rates of fault-revealing test cases (left) and mutation scores (right) w.r.t. N_{max} values.

Sasak's PPL could not handle some problems due to the absence of support for certain features of PDDL2.1 (column N/A) in that implementation. Second, the 7 optimal configurations of Fast Downward have been successfully tested with MorphinPlan without detecting any non-optimality behaviour. Regarding the known bug of PPL V0, it was revealed only on two problems from a single domain (elevator). It indicates that MorphinPlan shall be used with a sufficient variety of PDDL domain/problems as non-optimality bugs can sometimes only be revealed by a few domain definitions. Third, non-optimality behaviours were detected for all the 10 remaining configurations of Fast Downward (at least one 'Check' in all columns). This result is interesting, as the variety of problems, combined with the capability of MorphinPlan to explore the state space, leads to the systematic detection of existing non-optimal behaviours in AI planning. Regarding RQ3, the results of this second experiment show that MorphinPlan systematically detects non-optimal behaviours in the two selected AI planners.

6. Threats to Validity

As *internal threats-to-validity*, we identified the creation of mutated AI planners for guiding the MT framework and the choice of the maximum number of follow-up test cases. To cope with the first threat, we randomized the selection of mutants used by MorphinPlan and provided only average results, plus standard deviations. By doing this, our results show that MorphinPlan is agnostic to the usage of mutants to guide the test case selection. However, the 22 mutated planners were manually created by using a single idea (as explained in Section 4.3) and thus they may not be representative enough of the diversity of potential faults found in AI planning implementation. To cope with this threat, our second experiment dealt with off-the-shelf planners only, without any evaluation of the mutated AI planners. Regarding the maximum number of follow-up test cases, the results of Figure 4 indicate that the framework is dependent on N_{max} and the choice of this value may largely influence the results of any test case selection strategy. Even though more experiments are needed to further analyse this question, the preliminary results of Figure 4 are very encouraging.

As *external threats-to-validity*, we identified i) the selection of PDDL problems to perform our experiments, ii) the selection of planners to test and, iii) the applicability of MorphinPlan only to optimal AI planner. As explained in Section 4.2, our selection was objective but still, other PDDL domains and problems could lead to different results as nothing guarantees that our selection is representative enough of the huge diversity of AI planning problems. Also, our second experiment included only two AI planners in 20 distinct configurations. Even though these planners are off-the-shelf planners, the results could be different for other planners. By using a single MR dedicated to finding non-optimality issues, our approach focuses on testing optimal AI planning systems. Of course, other realistic planners, not necessarily seeking optimality but just validity of the generated plan, remain outside the scope of MorphinPlan. To cope with these threats and limitations, more experiments are needed to confirm the obtained results and the usage of more MRs are needed.

7. Related Work

The validation and verification of AI planning systems is an important issue that has been addressed from different angles since the last century [4]. Some approaches have focused on the testing of domain and problem definition, chasing faults in PDDL programs [26, 41, 22, 32]. A major approach for the verification of safety properties in domain specification is based on model-checking methods, especially for industrial application cases [37, 6, 46, 45, 1]. The general idea is to perform bounded model checking on the underlying state space associated with the specified PDDL domain. Interestingly, other approaches have formulated test case generation as AI planning problems [25, 5]. Our approach differs largely from these works in that it focuses on the testing of the AI planners for detecting non-optimal generated plans. Under the hypothesis that domain and problem specification are verified by other means, our contribution lies in the usage of mutation-guided MT for detecting optimality-related faults.

MT has been successful in testing general and autonomous software systems [44, 30, 29, 14, 2, 15, 49]. The essential contribution of these works lies in the definition of advanced MRs for complex learning systems. Unlike our approach which focuses on using one MR for checking the optimality of an AI planner, these works contribute to more general system testing without questioning the generation of follow-up test cases for AI planners. Recently, MT [18] was used to test learned action policy used in games and planning. The idea was to exploit state relaxation, an easy abstraction mechanism used in AI planning, to construct MRs.

The closest works regarding the usage of MT for checking optimality faults in AI planning systems are by Zhang et al. [54] and by Cheng et al. [12]. In [54], the authors propose to use the distance-related triangle inequality as an MR for testing graph-based path planning algorithms. Our MR defined based on cost-driven minimisation has similarities with this triangle inequality, but it differs from it by avoiding the necessary distance definition which is necessary for path planning. Also, the approach generalizes the usage of MT by working

directly at the level of PDDL for any domain definition (not only for path planning) which allows us to deal with a much larger class of problems. Note that MorphinPlan enables a dynamic exploration of the underlying search space without requiring its explicit construction. In [12], the goal is to test learnt policies for automated driving applications by using MT. The employed MR is based on mutating an input situation in such a way that optimality shall be preserved. Thus, it resembles our MR, which looks for derived states where optimality can easily be compared with the one provided by an initial planner run. However, our approach differs in its goal (testing a PDDL-based AI planning system) and approach (using mutated versions of a reference implementation). Unlike [54, 2, 55, 12], our approach introduces state selection strategies, including mutation adequacy selection, for generating the follow-up test cases. Despite the importance of recent findings in mutation testing [36, 48], to the best of our knowledge, it is the first attempt to use mutation testing for verifying AI planning systems. The difficulty of creating mutants for complex and optimized AI planners that exhibit optimality-related faults may explain the absence of such a tool in the literature.

8. Conclusion and Future Work

In this article, we have introduced a MT framework for testing optimality in AI planners with a mutation adequacy test selection strategy. Our tool MorphinPlan considers any PDDL planning domain and problem (written in a restricted fragment of PDDL2.1) and generates, based on the PUT's initial plan, new problems as follow-up test cases by exploring the state space of the problem. By comparing the actual results for these new problems to the cost of the initial plan, non-optimal AI planners can be detected using MT. In our framework, we score all the visited states with their ability to kill mutated AI planners, that have been previously created from a reference implementation, and we select the states having the greatest scores, i.e., the states most likely to reveal faults, to generate follow-up test cases. Our experimental evaluation shows that

MorphinPlan is effective at detecting non-optimal AI planners among both artificially created mutated AI planners and off-the-shelf, configurable planners. A key observation of our results is that mutation adequacy test case selection is well-tuned for detecting non-optimality and that it outperforms traditional breadth-first search and random strategies, provided that sufficient time can be allocated to the test generation and selection steps.

Our future work encompasses two directions. The first direction follows the recent research efforts in applying MT to learnt policies on Markov Decision Processes. For these models, only probabilistic AI planning approaches are relevant, as these processes are inherently non-deterministic. The second direction is the development of MT for AI planners based on PDDL3. In particular, we want to investigate MRs over the PDDL3 grammar (including domain definition and problem instances) for automating testing of PDDL planners.

References

[1] Mohammad Abdulaziz and Friedrich Kurz. 2023. Formally Verified SAT-Based AI Planning. In *Proc. of the AAAI Conference on Artificial Intelligence*. 14665–14673. https://doi.org/10.1609/aaai.v37i12.26714

[2] Jon Ayerdi, Pablo Valle, Sergio Segura, Aitor Arrieta, Goiuria Sagardui, and Maite Arratibel. 2023. Performance-Driven Metamorphic Testing of Cyber-Physical Systems. *IEEE Transactions on Reliability* 72, 2 (June 2023), 827–845. https://doi.org/10.1109/TR.2022.3193070

[3] Fahiem Bacchus. 2001. AIPS 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *Ai magazine* 22, 3 (2001), 47–47.

[4] S. Bensalem, K. Havelund, and A. Orlandini. 2014. Verification and validation meet planning and scheduling. *Int. Journal of Soft. Tools and Technology Transfer* 16 (2014), 1–12. https://doi.org/doi.org/10.1007/s10009-013-0294-x

[5] Josip Božić, Oliver Tazl, and Franz Wotawa. 2019. Chatbot Testing Using AI Planning. In *IEEE International Conference On Artificial Intelligence Testing, AITest 2019*. https://doi.org/10.1109/AITest.2019.00-10

[6] Guillaume P. Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Michael R. Lowry, Corina S. Pasareanu, Arnaud Venet, Willem Visser, and Richard Washington. 2004. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods Syst. Des.* 25, 2-3 (2004), 167–198. https://doi.org/10.1023/B:FORM.0000040027.28662.a4

[7] Mats Carlsson and Et al. 2019. *SICStus Prolog User's Manual, Release 4.5.1.*

[8] T.Y. Chen, S.C. Cheung, and S.M. Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST-CS98-01. Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

[9] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *Comput. Surveys* 51, 1 (2018), 1–27. https://doi.org/10.1145/3143561

[10] Tsong Yueh Chen and T. H. Tse. 2021. New Visions on Metamorphic Testing after a Quarter of a Century of Inception. In *Proc. of the 29th ACM Joint Meet. on European Soft. Eng. Conf. and Symp. on the Foundations of Soft. Eng. (ESEC/FSE), Aug. 23-28*. 1487–1490.

[11] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. 85–99.

[12] Mingfei Cheng, Yuan Zhou, Xiaofei Xie, Junjie Wang, Guozhu Meng, and Kairui Yang. 2024. Evaluating Decision Optimality of Autonomous Driving via Metamorphic Testing. arXiv:2402.18393 [cs.AI] https://arxiv.org/abs/2402.18393

[13] Stevo Alves de Andrade, Fatima L. S. Nunes, and Marcio Eduardo Delamaro. 2023. Exploiting deep reinforcement learning and metamorphic testing to automatically test virtual reality applications. *Software Testing, Verification and Reliability* (2023). https://doi.org/10.1002/stvr.1863 Published online Sep. 23rd.

[14] Yao Deng, Guannan Lou, Xi Zheng, Tianyi Zhang, Miryung Kim, Huai Liu, Chen Wang, and Tsong Yueh Chen. 2021. BMT: Behavior Driven Development-based Metamorphic Testing for Autonomous Driving Models. In *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*. 32–36. https://doi.org/10.1109/MET52542.2021.00012

[15] Yao Deng, Xi Zheng, Tianyi Zhang, Huai Liu, Guannan Lou, Miryung Kim, and Tsong Yueh Chen. 2022. A Declarative Metamorphic Testing Framework for Autonomous Driving. *IEEE Transactions on Software Engineering* (2022). https://doi.org/10.1109/TSE.2022.3206427

[16] Yannis Dimopoulos, Alfonso Gerevini, Patrik Haslum, and Alessandro Saetti. 2006. The benchmark domains of the deterministic part of IPC-5. *Abstract Booklet of the competing planners of ICAPS-06* (2006), 14–19.

[17] Yefim Dinitz and Shay Solomon. 2007. On Optimal Solutions for the Bottleneck Tower of Hanoi Problem. In *Int. Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM) (LNCS, Vol. 4362)*. 248–259.

[18] Hasan Ferit Eniser, Timo P. Gros, Valentin Wüstholz, Jörg Hoffmann, and Maria Christakis. 2022. Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing. In *Proc. of the 31st*

ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA). https://doi.org/10.1145/3533767.3534392

[19] Giovani Farias, Ramon Fraga Pereira, Lucas Hilgert, Felipe Meneguzzi, Renata Vierira, and Rafael Bordini. 2017. Predicting Plan Failure by Monitoring Action Sequences and Duration. *ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal* 6, 2 (2017).

[20] Malik Ghallab, Dana Nau, and Paolo Traverso. 2016. *Automated Planning and Acting*. Cambridge University Press. https://projects.laas.fr/planning/book.pdf

[21] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. 2019. An Introduction to the Planning Domain Definition Language. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 13 (04 2019), 1–187. https://doi.org/10.2200/S00900ED2V01Y201902AIM042

[22] Klaus Havelund, Alex Groce, Gerard Holzmann, Rajeev Joshi, and Margaret Smith. 2009. Automated testing of planning models. *Lecture Notes in Computer Science* 5348 LNAI (2009). 5th Int. Workshop on Model Checking and Art. Intel., MoChArt 2008.

[23] Malte Helmert. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26 (2006), 191–246.

[24] Jörg Hoffmann and Stefan Edelkamp. 2005. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research* 24 (2005), 519–579.

[25] Adele Howe, Anneliese Mayrhauser, and Richard Mraz. 1997. Test Case Generation as an AI Planning Problem. *Automated Software Engineering* 4 (Jan. 1997), 77–106.

[26] R. Howey and D. Long. 2003. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In

Proceedings of the ICAPS 2003 workshop on "The Competition: Impact, Organization, Evaluation, Benchmarks".

[27] Casidhe Hutchison, Milda Zizyte, Patrick E. Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. 2018. Robustness Testing of Autonomy Software. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), Gothenburg, Sweden*. 276–285.

[28] Upulee Kanewala, James M. Bieman, and Asa Ben-Hur. 2016. Predicting Metamorphic Relations for Testing Scientific Software: A Machine Learning Approach Using Graph Kernels. *Software Testing, Verification and Reliability* 26, 3 (May 2016), 245–269. https://doi.org/10.1002/stvr.1594

[29] Rui Li, Huai Liu, Guannan Lou, Xi Zheng, Xiao Liu, and Tsong Yueh Chen. 2021. Metamorphic Testing on Multi-module UAV Systems. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE51524.2021.9678841

[30] Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. 2017. Metamorphic Model-Based Testing of Autonomous Systems. *2nd International Workshop on Metamorphic Testing, MET 2017* (2017). https://doi.org/10.1109/MET.2017.6

[31] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* (2014). https://doi.org/10.1109/TSE.2013.44

[32] Maurício C. Magnaguagno, Ramon Fraga Pereira, Martin D. Móre, and Felipe Meneguzzi. 2020. *Web Planner: A Tool to Develop, Visualize, and Test Classical Planning Domains. In Book Knowledge Engineering Tools and Techniques for AI Planning*. Springer, Chapter 11, 209–227. https://doi.org/10.1007/978-3-030-38561-3_11

[33] Drew M McDermott. 2000. The 1998 AI planning systems competition. *AI magazine* 21, 2 (2000), 35–35.

[34] Christian Muise. 2016. Planning.domains. In *Proc. of Int. Conf. on Automated Planning and Scheduling. System Demo. Track. London, UK. June 12-17*. 242–250.

[35] IPC 2008 Organizers. 2008. IPC 2008 Website. `https://ipc08.icaps-conference.org/deterministic/index.html`

[36] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship between Mutants and Real Faults. In *Proc. of the 40th Int. Conf. on Software Engineering (ICSE)*. `https://doi.org/10.1145/3180155.3180183`

[37] John Penix, Charles Pecheur, and Klaus Havelund. 1999. Using Model Checking to Validate AI Planner Domain Models. In *Proc. of the 23rd Annual Software Engineering Workshop, NASA Goddard*. `https://ntrs.nasa.gov/citations/19990054671`

[38] Giuseppe Penna, Daniele Magazzeni, Fabio Mercorio, and Intrigila Benedetto. 2009. UPMurphi: A tool for universal planning on PDDL+ problems. In *Proc. of the 19th Int. Conf. on Automated Planning and Scheduling (ICAPS'09). September 19-23, Thessaloniki, Greece.*

[39] AIPlan4EU H2020 European Project. 2021. `https://www.aiplan4eu-project.eu`

[40] Franco Raimondi, Charles Pecheur, and Guillaume Brat. 2007. Testing Planning Domains (without Model Checkers). *Electron. Notes Theor. Comput. Sci.* 190, 2 (2007).

[41] Franco Raimondi, Charles Pecheur, and Guillaume Brat. 2009. PDVer, a Tool to Verify PDDL Planning Domains. In *Proc. of ICAPS'09 Work-*

shop on Verification and Validation of Planning and Scheduling Systems. September 19-23, Thessaloniki, Greece.

[42] Robert Sasak. 2021. Prolog Planning Library. `https://github.com/RobertSasak/Prolog-Planning-Library`.

[43] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortes. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824. `https://doi.org/10.1109/TSE.2016.2532875`

[44] Sergio Segura, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. 2020. Metamorphic Testing: Testing the Untestable. *IEEE Software* 37, 3 (2020), 46–53. `https://doi.org/10.1109/MS.2018.2875968`

[45] Anas Shrinah and Kerstin Eder. 2020. Goal-constrained planning domain model verification of safety properties. In *Proceedings of the 9th European Starting AI Researchers' Symposium 2020 co-located with 24th European Conference on Artificial Intelligence (ECAI 2020) (CEUR Workshop Proceedings, Vol. 2655).*

[46] Margaret H Smith, Gerard J Holzmann, Gordon C Cucullu, and BD Smith. 2005. Model Checking Autonomous Planners: Even the Best Laid Plans Must be Verified. In *Proc. of the IEEE Aerospace Conference. Big Sky, Montana. In March.*

[47] Helge Spieker and Arnaud Gotlieb. 2020. Adaptive metamorphic testing with contextual bandits. *Journal of Systems and Software* 165 (2020). `https://doi.org/10.1016/j.jss.2020.110574`

[48] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. 2020. Selecting Fault Revealing Mutants. *Empirical Software Engineering* 25, 1 (2020), 434–487. `https://doi.org/10.1007/s10664-019-09778-7`

[49] Dongwei Xiao, Zhibo LIU, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, Vol. 6. Association for Computing Machinery, New York, NY, USA, Article 15, 28 pages. https://doi.org/10.1145/3508035

[50] Xiaoyuan Xie, Joshua Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tson Chen. 2009. Application of Metamorphic Testing to Supervised Classifiers. In *2009 Ninth Int. Conf. on Quality Software*, Vol. 33. 135–144.

[51] Xiaoyuan Xie, Joshua W.K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and Validating Machine Learning Classifiers by Metamorphic Testing. *Journal of Systems and Software* 84, 4 (2011), 544–558. https://doi.org/10.1016/j.jss.2010.11.920

[52] Liming Xu, Dave Towey, Andrew P. French, Steve Benford, Zhi Quan Zhou, and Tsong Yueh Chen. 2018. Enhancing Supervised Classifications with Metamorphic Relations. *Proceedings of the 3rd International Workshop on Metamorphic Testing - MET '18* (2018), 46–53. https://doi.org/10.1145/3193977.3193978

[53] Shin Yoo. 2010. Metamorphic Testing of Stochastic Optimisation. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 192–201. https://doi.org/10.1109/ICSTW.2010.26

[54] Jiantao Zhang, Zheng Zheng, Beibei Yin, Kun Qiu, and Yang Liu. 2019. Testing Graph Searching Based Path Planning Algorithms by Metamorphic Testing. In *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*. 158–15809. https://doi.org/10.1109/PRDC47002.2019.00046

[55] Xiao-Yi Zhang, Yang Liu, Paolo Arcaini, Mingyue Jiang, and Zheng Zheng. 2024. MET-MAPF: A Metamorphic Testing Approach for Multi-Agent Path Finding Algorithms. *ACM Trans. Softw. Eng. Methodol.* (jun 2024). https://doi.org/10.1145/3669663 Just Accepted.

[56] Zhi Quan Zhou and Liqun Sun. 2019. Metamorphic Testing of Driverless Cars. *Commun. ACM* 62, 3 (2019), 61–67. https://doi.org/10.1145/3241979

Definition	Description
$f_1(s) = h(s)$	Expands the best states first.
$f_2(s) = -h(s)$	Expands the worst states first.
$f_3(s) = -g(s) + h(s)$	Encourages the search to expand the deepest nodes. It uses $h(s)$ to mitigate its naive trend to always search for a longer solution, thus leading to a timeout signal.
$f_4(s) = (-1)^{g(s)}h(s)$	Scores the discovered state with h but "discards" it by signing it with the parity of its cost.
$f_5(s) = -g(s) + (-1)^{g(s)}h(s)$	Combination of f_3 and f_4.
$h_1(s) = \|s\|$	Ranks a state by its length (i.e, its number of true facts).
$h_2(s) = \|g \setminus (s \cap g)\|$	Ranks a state with the number of facts of the goal that are not true yet.
$h_3(s) = s\Delta i$	Ranks a state with its distance to the initial state. The distance used is the symmetric difference.
$h_4(s) = s\Delta g$	Ranks a state with its distance to the goal state.
$h_5(s) = \|succ(s)\|$	Ranks a state with its number of applicable actions. If we suppose that every action leads to a distinct state, then h_5 can be defined as the number of successors of the current state.
$h_6(s) = h_{max}$	Ranks a state with the index of the first fact layer of the relaxed planning graph, which contains all the facts of the goal. In our planning setting, where all action costs equal 1, it corresponds to the classical heuristic h_{max}.

Table 2: Description of the f and h mutated functions used in a flawed A^* to create the mutated AI planners. $g(s)$ refers to the cost of the plan used to reach the state s. g and i refer to the goal and the initial state, respectively.

	$f_1(s)$	$f_2(s)$	$f_3(s)$	$f_4(s)$	$f_5(s)$
$h_1(s)$	✓	✓	✓	✓	✓
$h_2(s)$	✓	✓	✓	E	E
$h_3(s)$	E	✓	✓	✓	✓
$h_4(s)$	E	✓	✓	E	✓
$h_5(s)$	✓	✓	✓	✓	✓
$h_6(s)$	E	✓	✓	E	E

Table 3: Overview of mutant combinations of f and h: ✓ indicates the used mutants, E marks equivalent mutants (unused).

	MorphinPlan		bfs_det		bfs_ran		wal_ran	
	Mut. Score	t (s)	Mut. Score	t (s)	Mut. Score	t (s)	Mut. Score	t (s)
airport06	**83.5±7.7**	175.8	25.3±7.5	17.9	79.6±5.4**	18.0	60.8±18.7***	2.8
airport07	**83.1±6.8**	176.2	13.6±5.9	13.3	75.3±8.2**	13.3	59.9±11.4***	1.6
blocks01	**82.2±6.9**	23.0	68.6±7.0	0.1	30.2±8.3*	0.1	18.2±7.9**	0.9
blocks02	**94.5±3.9**	23.0	66.5±8.7	0.1	74.6±7.2*	0.1	34.8±9.1**	1.2
blocks03	**83.9±8.9**	22.7	55.9±7.4	0.1	57.1±11.2**	0.1	32.8±8.9**	1.0
gripper01	**76.6±10.9**	29.9	33.0±9.6	0.1	70.3±5.8**	0.2	22.7±7.5**	2.4
elevator02	**92.5±8.2**	20.7	66.4±6.3	0.2	78.6±8.7*	0.1	3.9±4.6*	1.1
elevator03	**94.2±8.4**	29.4	55.7±8.6	0.1	70.9±7.1**	0.5	23.1±6.2**	2.1
openstacks01	**27.4±9.9**	213.9	12.7±8.1	0.8	17.2±6.1**	0.8	22.9±6.0**	0.2
pegsol04	**90.8±7.4**	45.8	72.6±7.8	3.4	45.2±15.5**	3.3	42.2±13.2**	1.0
pegsol05	**86.2±6.9**	51.3	33.6±7.7	4.7	46.5±24.1**	4.7	23.7±12.6***	1.1
pegsol06	**98.6±6.1**	147.5	29.1±8.5	6.9	45.0±12.7**	6.9	48.7±10.2***	1.6
psr-small03	**83.3±8.6**	22.7	41.8±9.1	0.1	41.3±11.0**	0.1	15.3±8.2**	0.5
psr-small06	**94.4±4.7**	27.4	53.5±5.7	0.2	76.0±9.7*	0.2	51.6±8.7**	0.9
psr-small08	**76.3±6.7**	27.5	56.1±7.9	0.2	48.8±10.1**	0.1	30.7±8.4**	0.8
psr-small09	**100.0±0.0**	26.9	84.0±5.6	0.2	57.5±10.2**	0.2	37.4±9.7**	1.1
tpp03	**75.8±9.4**	23.0	16.1±7.7	0.2	46.3±10.8**	0.2	18.2±8.5**	104.5
transport01	**96.0±5.3**	28.8	83.2±9.0	0.2	81.3±8.2*	0.3	49.9±9.1**	1.7
travel02	**89.5±4.4**	22.2	58.2±10.2	0.5	49.4±9.5**	0.5	49.6±5.9**	4.6
Mean±std. dev.	**84.7±15.3**	59.9	48.7±22.2	2.6	57.4±18.0	2.6	34.0±15.8	6.9

Table 4: Average mutation scores of MorphinPlan on the 19 problems for the 20 subsets of validation/selection mutants. For the non-deterministic methods, the standard deviation over 10 runs is indicated (* $\leq$ 5, ** $\leq$ 10, *** $\leq$ 15).

	Sasak's PPL [42]			FastDownward						
		V1		Heuristic (Non-Optimal) Configurations						
	V0	A_{h0}	A_{hmax}	A_{ff}	$A_{add,cea,cg}$	W_{ff}	$W_{add,cea}$	W_{cg}	$W_{goalcount}$	Opt.
blocks01	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
blocks02	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
blocks03	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
elevator02	✓	✗	✗	✗	✓	✗	✓	✓	✗	✗
elevator03	✓	✗	✗	✗	✓	✓	✓	✓	✓	✗
psr-small03	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
psr-small06	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
psr-small08	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
psr-small09	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
airport06	N/A	✗	N/A	✗	✗	✗	✗	✗	✗	✗
airport07	N/A	✗	N/A	✗	✗	✓	✗	✓	✗	✗
pegsol04	N/A	✗	N/A	✗	✓	✓	✓	✓	✓	✗
pegsol05	N/A	✗	N/A	✗	✓	✓	✓	✓	✓	✗
pegsol06	N/A	✗	N/A	✗	✓	✓	✓	✓	✓	✗
gripper01	N/A	N/A	N/A	✗	✓	✓	✓	✓	✓	✗
openstacks01	N/A	N/A	N/A	✓	✓	✓	✓	✓	✗	✗
tpp03	N/A	N/A	N/A	✗	✓	✗	✓	✓	✓	✗
transport01	N/A	N/A	N/A	✗	✗	✗	✗	✗	✓	✗
travel02	N/A	N/A	N/A	✗	✗	✗	✗	✗	✗	✗

Table 5: MorphinPlan results of non-optimal plans detection on 20 configs of two off-the-shelf AI planners. Columns showing similar behaviour have been merged. Checks indicate problems which have led to at least one non-optimal plan. Crosses indicate that no such plans were detected. N/A are cases where the planner failed due to unsupported PDDL features.