Automatic Metamorphic Test Oracles for Action-Policy Testing

Jan Eisenhut[1], **Álvaro Torralba**[2], **Maria Christakis**[3], **Jörg Hoffmann**[1,4]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, ⟨lastname⟩@cs.uni-saarland.de
[2] Aalborg University, Denmark, alto@cs.aau.dk
[3] TU Wien, Austria, maria.christakis@tuwien.ac.at
[4] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany

Abstract

Testing is a promising way to gain trust in learned action policies π. Prior work on action-policy testing in AI planning formalized bugs as states t where π is sub-optimal with respect to a given testing objective. Deciding whether or not t is a bug is as hard as (optimal) planning itself. How can we design *test oracles* able to recognize some states t to be bugs *efficiently*? Recent work introduced *metamorphic* oracles which compare policy behavior on state pairs (s, t) where t is easier to solve; if π performs worse on t than on s, we know that t is a bug. Here, we show how to automatically design such oracles in classical planning, based on simulation relations between states. We introduce two oracle families of this kind: first, morphing query states t to obtain suitable s; second, maintaining and comparing upper bounds on h^* across the states encountered during testing. Our experiments on ASNet policies show that these oracles can find bugs much more quickly than the existing alternatives, which are search-based; and that the combination of our oracles with search-based ones almost consistently dominates all other oracles.

1 Introduction

Learned action policies π represented by neural networks are gaining traction in AI planning (Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020; Karia and Srivastava 2021; Stahlberg, Bonet, and Geffner 2022). Once learned, action policies can be used to make real-time decisions in dynamic environments. Yet, this raises obvious concerns regarding potential policy "bugs", that is, undesirable or even fatal policy behavior in particular situations. Testing – searching for bugs – is a natural paradigm to address these concerns. Specifically, testing can serve to assess the quality of π, and when used extensively, to certify that π can be trusted.

There has been substantial work on policy testing in sequential decision making. Most of it focuses on so-called stress testing, which tries to find environment behavior under which a policy fails (e.g., Dreossi et al. 2015; Akazaki et al. 2018; Koren et al. 2018; Ernst et al. 2019; Lee et al. 2020). But if the failure is unavoidable, then π is not actually doing anything wrong, and the situation hardly qualifies as a "bug". Recent work (Steinmetz et al. 2022) addressed this issue in the context of AI planning, defining *bugs* as states where π is sub-optimal with respect to a given *testing objective*. Here, we address classical planning with the testing objective of *additive-cost minimization*. In this context, a state t is a bug in π if the cost of π's plan, which we denote by $c^\pi(t)$, is greater than that of an optimal plan $h^*(t)$.

Deciding whether t is a bug subsumes cost-optimal planning. How can we design *test oracles* able to recognize some t to be bugs *efficiently*? Steinmetz et al. (2022) experiment with plan-improvement oracles as well as simple oracles leveraging invertible actions. Here, we address a new class of oracles based on recent ideas from *metamorphic testing*.

Metamorphic testing is a well established method for designing oracles in software testing (Chen, Cheung, and Yiu 1998). The idea is to test program behavior on inputs chosen such that it is known how the respective outputs should relate. Eniser et al. (2022) transfer this idea to action-policy testing by using a notion of relaxations. The intuition is simple: if t relaxes s, i.e., t is easier to solve than s, yet π performs worse on t than on s, then we know that t is a bug in π. Eniser et al. apply this approach on three Atari-like games, with manually designed relaxation relations that are based on making obstacles easier to avoid. Here, we instead show how to *automatically* design such oracles, in the planning context, based on *simulation* relations between states.

A simulation (Milner 1971; Gentilini, Piazza, and Policriti 2003) is a relation $\preceq$ on states where, whenever $s \preceq t$, for every transition $s \xrightarrow{a} s'$ there exists a transition $t \xrightarrow{a} t'$ such that $s' \preceq t'$. In words, t simulates s if anything we can do in s, we can also do in t, leading to a simulating state. Obviously (and as Eniser et al. point out), this kind of relation qualifies for their oracles. Our contribution is the practical realization and evaluation of this idea. We leverage prior work (Torralba and Hoffmann 2015; Torralba 2017, 2018) on automatically extracting simulations in planning. Precisely, we leverage the more general notion of *dominance functions* (Torralba 2017), which are *quantitative*: they say by *how much* (plan cost) t is easier than s. This allows us to generalize Eniser et al.'s approach. If t is not easier but *harder* than s, yet only by cost ≤ 2, then we can still conclude that t is a bug if π performs worse on t by cost > 2.

We introduce two families of oracles based on these ideas. Our *state-morphing oracles* follow the design by Eniser et al. (2022), taking an input state t, generating morphed states s, and reporting t to be a bug if the comparison of π's behavior on any s allows to do so. The main technical issue we address here is how to generate the morphed states.

Our second oracle family, *bound-maintenance oracles*, instead maintains upper bounds $u(t)$ on plan cost across the states t encountered during testing, explicitly leveraging the ability of quantitative simulation relations to propagate such bounds. The initial source of upper bounds are policy runs. Whenever a new or better bound $u(t)$ is found, all neighbors of t in the simulation relation can be updated too. Moreover, search methods – limited lookahead search or search-based external test oracles (running either a plan-improvement algorithm or an actual planner) – can be naturally plugged in as another initial source of upper bounds. This results in synergy between information provided by search vs. simulation, going beyond pure metamorphic testing.

We evaluate our test oracles on action policies learned with ASNets (Toyer et al. 2018, 2020). Specifically, we use the collection of benchmarks and policies introduced by Steinmetz et al. (2022). We compare against Steinmetz et al.'s test oracles as well as other search-based oracles. Our key findings are that our new metamorphic oracles find bugs much more quickly than search-based ones; and that our combination of metamorphic and search-based oracles almost consistently dominates all other oracles.

2 Background

We next briefly give the necessary background encompassing the planning framework, simulation relations in planning, and the relevant prior work on test oracles (Steinmetz et al. 2022; Eniser et al. 2022).

2.1 Finite-Domain Representation Planning

An **FDR task** is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, c, I, G\rangle$. $\mathcal{V}$ is a finite set of **variables**; each $v \in \mathcal{V}$ is associated with a finite domain $dom(v)$. A **state** s is a complete variable assignment; we denote the set of all states by $\mathcal{S}$. $\mathcal{A}$ is a finite set of **actions**; each $a \in \mathcal{A}$ is a pair $(\mathsf{pre}(a), \mathsf{eff}(a))$ of **precondition** and **effect**, both partial assignments. c is a cost function $c\colon \mathcal{A} \to \mathbb{R}^{0+}$. $I \in \mathcal{S}$ is the **initial state**. The **goal** G is a partial assignment.

Action a is **applicable** in state s if $\mathsf{pre}(a) \subseteq s$; we denote the set of actions applicable to s by $A[s]$. Applying $a \in A[s]$ to s changes the value of the variables affected by $\mathsf{eff}(a)$ to $\mathsf{eff}(a)[v]$ and leaves s unchanged elsewhere. $s[\![a]\!]$ denotes the resulting state after applying a, and similarly, $s[\![\vec{a}]\!]$ after applying an action sequence. $\vec{a}$ is a **plan** for s if $G \subseteq s[\![\vec{a}]\!]$. We denote the summed-up cost of $\vec{a}$ by $c(\vec{a})$. A plan $\vec{a}$ is **optimal** for s if its cost $c(\vec{a})$ is minimal among all plans for s. We denote by $h^*(s)$ the exact **cost-to-goal**, i.e., the cost of an optimal plan for s if one exists and ∞ otherwise.

2.2 Dominance Functions in FDR

Simulation relations (Milner 1971) aim to identify whether a system can mimic all behaviors of another. In the context of planning, this means that whenever $s \preceq t$ any plan for s is also a plan for t. When only cost-to-goal is of interest, this can be specialized into dominance relations (Torralba and Hoffmann 2015), which only require t to have a plan of lower or equal cost, possibly using fewer and/or completely different actions. Formally, a relation $\preceq \subseteq \mathcal{S} \times \mathcal{S}$ is a **dominance relation** if $s \preceq t$ implies $h^*(t) \leq h^*(s)$. Here, we build on dominance functions (Torralba 2017):

Definition 1. A **dominance function** is a function $\mathcal{D}\colon \mathcal{S} \times \mathcal{S} \to \mathbb{R} \cup \{-\infty\}$ such that $\mathcal{D}(s,t) \leq h^*(s) - h^*(t)$ for all $s \in \mathcal{S}$ with $h^*(s) < \infty$ and $t \in \mathcal{S}$.

Dominance functions directly generalize dominance relations, providing information about their relative cost-to-goal. Whenever $\mathcal{D}(s,t) \geq 0$, t dominates/relaxes s; the value indicates *how much* easier t is than s. If $-\infty < \mathcal{D}(s,t) < 0$, s can be harder than t but at most by cost $|\mathcal{D}(s,t)|$. Finally, if $\mathcal{D}(s,t) = -\infty$, it means the function does not give a bound here.

Dominance functions can be automatically derived in a compositional manner. First, the task Π is divided into a set of factors through partitioning the set of variables $\mathcal{V}$ into k disjoint subsets $V_1, \dots, V_k$. For any state s, we denote by s_i the projection of s onto V_i. We construct a factor $\Theta_1, \dots, \Theta_k$ as the projection of Π onto each V_i. Then, a separate dominance function $\mathcal{D}_i$ is computed for each factor, such that for any pair of states s, t, $\sum_{i=1}^{k} \mathcal{D}_i(s_i, t_i) \leq h^*(s) - h^*(t)$. As detailed by Torralba (2017), this can be done by a fixpoint computation akin to that of simulation relations, in time polynomial in the size of the factors. This procedure is run once for Π, as a pre-process. Later, given two states s and t, we simply compute $\mathcal{D}(s,t) = \sum_{i=1}^{k} \mathcal{D}_i(s_i, t_i)$.

2.3 Policy Testing

We consider deterministic memoryless policies for FDR tasks. Hence, a **policy** is a function $\pi\colon \mathcal{S} \to \mathcal{A}$ that maps states to applicable actions $\pi(s) \in A[s]$. We denote the unique **run of** π **on** s – the action sequence resulting from iteratively applying π starting from s – by $\sigma^\pi(s)$. Note that learning a policy for FDR tasks is useful if (like ASNets) the policy generalizes over all instances of a domain.

Given a state s, we denote the cost of π's plan as follows:

$$c^\pi(s) := \begin{cases} c(\sigma^\pi(s)) & \sigma^\pi(s) \text{ is a plan} \\ \infty & \text{otherwise} \end{cases}$$

Focusing on additive-cost minimization as our testing objective – i.e., inserting this standard objective into Steinmetz et al.'s (2022) general definition – we say that a state t is a **bug** if $c^\pi(t) > h^*(t)$. Note that this is the case if either π does not find a plan although t is solvable (and hence $h^*(t) < \infty$); or π does find a plan but its cost is not optimal.[1]

Policy testing consists of two separate activities: (a) generating test states t; and (b) running test oracles on these t to find bugs. Here, we are exclusively concerned with (b). For (a), we adopt Steinmetz et al.'s techniques, which use simple random-walk methods to iteratively build up a **pool** $\mathcal{P}$ of test states $t \in \mathcal{P}$.[2] Steps (a) and (b) can be interleaved, i.e., the oracles do not assume that $\mathcal{P}$ has already been completed.

[1]Steinmetz et al. (2022) distinguish these two kinds of bugs. Here, this is not necessary as both are special cases of additive-cost minimization and are handled using the same techniques.

[2]Steinmetz et al. consider the design of biases towards states where π performs badly. As these biases have side effects on oracle

Steinmetz et al. run two simple kinds of oracles. First, **plan-improvement oracles** that report t to be a bug if they succeed in improving the plan generated by π. Specifically, Steinmetz et al. run **Aras** (Nakhost and Müller 2010) as well as a simple **look+policy** method consisting of depth-first search (up to depth 2) and running π on every leaf state of that search. Second, Steinmetz et al. consider the special case of invertible actions (e.g. (Daum et al. 2016)). There, all states are solvable so, if π does not find a plan for t, then t is a bug; and we can find a plan for cost comparison by going back to the initial state and using π from there. This yields **invertibility oracles** that apply only to the special case.

The oracles we introduce here are based on ideas from *metamorphic testing* (Chen, Cheung, and Yiu 1998). In software engineering, metamorphic testing essentially works by transforming a program input s.t. the new program output has an a-priori known relationship to the original output. In the context of action policies, Eniser et al. (2022) used a concept of *state relaxations* $\preceq$ to obtain metamorphic oracles, as follows: if $s \preceq t$, meaning that t is easier than s, and π performs worse on t than on s, then t is a bug in π. In our context: if $s \preceq t$ and $c^\pi(s) < c^\pi(t)$ then t is a bug in π. Eniser et al. perform case studies on three 2D-world single agent games, with manually designed relaxations modifying (fixed or moving) obstacles. They design oracles based on *unrelaxing* a test state $t \in \mathcal{P}$ into a morphed state s, reporting t to be a bug if π performs better on s.

Here, we automate the design of such oracles in the planning context. We introduce two families of oracles based on dominance functions.

3 State-Morphing Oracles

State-morphing oracles are based on comparing each pool state t against a number of **morphed** states s, obtained by modifying t in a manner guided by the dominance function.

3.1 The Principle

First, we introduce a bug criterion based on dominance functions. Assume that t dominates s by value $\mathcal{D}(s,t)$. If π performs optimally on t, then $c^\pi(t)$ should be at least $\mathcal{D}(s,t)$ lower than $c^\pi(s)$. Otherwise, t must be a bug.

Proposition 2 (Bug Criterion)**.** Let $\mathcal{D}$ be a dominance function and $s,t \in \mathcal{S}$ such that $c^\pi(s) < \infty$ and $\mathcal{D}(s,t) > -\infty$. If $c^\pi(s) - c^\pi(t) < \mathcal{D}(s,t)$, then t is a bug in π.

Proof. Because of $c^\pi(s) < \infty$, and thus $h^*(s) < \infty$, we have that $\mathcal{D}(s,t) \leq h^*(s) - h^*(t)$. As $-\infty < \mathcal{D}(s,t)$, this implies that $-\infty < h^*(s) - h^*(t)$, and hence, due to $h^*(s) < \infty$, that $h^*(t) < \infty$. Since therefore $c^\pi(s), h^*(s), h^*(t) < \infty$, $c^\pi(s) - c^\pi(t) < \mathcal{D}(s,t) \leq h^*(s) - h^*(t)$ implies that $0 \leq c^\pi(s) - h^*(s) < c^\pi(t) - h^*(t)$, so that $c^\pi(t) > h^*(t)$. □

This generalizes the bug criterion used by Eniser et al. (2022). A relaxation relation $\preceq$ can be expressed as a dominance function where $\mathcal{D}(s,t) = 0$ if $s \preceq t$, and $\mathcal{D}(s,t) = -\infty$ otherwise. Then, whenever $\mathcal{D}(s,t) = 0$, s unrelaxes t in the sense that $h^*(t) \leq h^*(s)$, so the policy should perform at least as well on t as it does on s. However, dominance functions have two additional capabilities that allow us to identify more bugs.

First, if $\mathcal{D}(s,t) > 0$, we know that t is *strictly* easier, $h^*(t) < h^*(s)$. We can then strengthen our comparison, requiring that π performs *better* on t by at least $\mathcal{D}(s,t)$, detecting bugs in particular even where $c^\pi(t) = c^\pi(s)$.

Second, we no longer even require that s unrelaxes t. If $-\infty < \mathcal{D}(s,t) < 0$, then s may be closer to the goal than t, but not by more than $|\mathcal{D}(s,t)|$. Consider for example that s is morphed from t by *removing* an obstacle (rather than adding one); but the obstacle could be avoided anyway by spending C units of cost, so $\mathcal{D}(s,t) = -C$. Then, if π performs worse on t than on s by a margin $> C$, we can report t to be a bug because avoiding the obstacle would result in a better plan.

Both capabilities may significantly increase the number of morphed states s we can compare to our pool state $t \in \mathcal{P}$.

Algorithm 1: State-morphing oracle

```
1 Procedure SMO($t$)
2     for $s \in$ morphState($t$) do
3         if $c^\pi(t) < \infty$ then $c_{cutoff} \leftarrow c^\pi(t) + \mathcal{D}(s,t)$;
4         else $c_{cutoff} \leftarrow \infty$;
5         $c_s \leftarrow$ run($\pi, s, c_{cutoff}$);            // $c^\pi(s) \leq c_s$
6         if $c_s < \infty \wedge c_s - c^\pi(t) < \mathcal{D}(s,t)$ then
7             flagAsBug($t$);
8             break;
```

Algorithm 1 shows the pseudo-code of our state-morphing oracle. The selection of morphed states to which we compare t is encapsulated in the morphState function and discussed in the next subsection. To compare the pool state t to each morphed state s in the SMO procedure, we need to run π on both states. Here, and everywhere below, we **cache** $c^\pi(s)$ for all states s on which π was already run. This is important for policies, like the ASNets (Toyer et al. 2020) we experiment with, that are slow to evaluate, making policy runs a major bottleneck in testing.

Furthermore, while Algorithm 1 determines $c^\pi(t)$ exactly, we only run π on the morphed states s as long as $c^\pi(s)$ is low enough to confirm t as a bug under our bug criterion. If $c^\pi(t) < \infty$, then $c^\pi(s) - c^\pi(t) < \mathcal{D}(s,t)$ is equivalent to $c^\pi(s) < \mathcal{D}(s,t) + c^\pi(t)$ so that the criterion cannot be met if $c^\pi(s) \geq c_{cutoff} = \mathcal{D}(s,t) + c^\pi(t)$. This means that if $\mathcal{D}(s,t) + c^\pi(t)$ is not positive, running π on s can be skipped altogether. In the case that $c^\pi(t) = \infty$, there cannot be such a cutoff bound since every plan found for s, no matter how long, is sufficient to prove that t is a bug.

We handle the partial evaluation of π on s in the run(π, s, c_{cutoff}) call, which returns an upper bound c_s on $c^\pi(s)$. In particular, it returns ∞ if the accumulated cost of the actions selected by π reach c_{cutoff}, or π visits a state recognized to be a dead end by $h^{\max}$. When checking the bug criterion, we replace $c^\pi(s)$ with c_s. Since $c_s - c^\pi(t) <$

performance, we do not employ them here, using instead Steinmetz et al.'s baseline with uniform-random action selection.

$\mathcal{D}(s,t)$ is implied by $c^\pi(s) - c^\pi(t) < \mathcal{D}(s,t)$, any state reported as a bug is indeed a bug by Proposition 2.

3.2 Morphing States

The performance of state-morphing oracles is highly influenced by how we morph t in order to derive states s for comparison. This can be done in many possible ways, and in particular, we do not necessitate that the morphed states are reachable from the initial state. A good morphing procedure should provide states s that are likely to show t to be a bug. In particular, they should always satisfy $\mathcal{D}(s,t) > -\infty$.

Naïvely, one could repeatedly guess a state s (e.g. at random) until one is found where $\mathcal{D}(s,t) > -\infty$. However, it is of course an advantage to do this more systematically by taking into account the dominance function $\mathcal{D}$. Hence our Algorithm 2 considers all states s that differ from t in exactly one state variable, and returns a random selection among those that fulfill $\mathcal{D}(s,t) > -\infty$.

Algorithm 2: The morphState function

```
1 Function morphState($t$)
2     $S \leftarrow \emptyset$ ; // set of states $s \neq t$ with $\mathcal{D}(s,t) > -\infty$
3     for $v \in \mathcal{V}$ do
4         for $d \in dom(v)$ do
5             if $d = t[v]$ then continue;
6             $s \leftarrow t$;
7             $s[v] \leftarrow d$;
8             if $\mathcal{D}(s,t) > -\infty$ then $S \leftarrow S \cup \{s\}$;
9     return select($S$, maxsize);
```

This simple procedure is obviously fast. Regarding its power to identify suitable morphed states, observe that states close to t are more likely to satisfy $\mathcal{D}(s,t) > -\infty$. To understand why, consider again how dominance functions are computed. The dominance function is given as a sum over factors $\mathcal{D}(s,t) = \sum_i \mathcal{D}_i(s_i, t_i)$. So we must have $\mathcal{D}_i(s_i,t_i) > -\infty$ for all factors. Modifying only a single variable maximizes the chances for this to be satisfied.

Observe that, if the factors are individual variables, then *Algorithm 2 will return at least one morphed state so long as any state s with $\mathcal{D}(s,t) > -\infty$ exists*, since $\mathcal{D}(s,t) > -\infty$ then arises from a single variable already. For general factors, the same would hold when changing Algorithm 2 to operate on the factor level. Yet, that would greatly increase runtime (iterating over variable-value tuples rather than single values). Moreover, with slow policies like ASNets, checking a larger number of morphed states is infeasible anyway, which is also the reason for the *maxsize* parameter.

4 Bound-Maintenance Oracles

We now introduce a new class of oracles, which propagates bounding information across the states encountered during testing and works much better in practice. We start with the motivation and basic idea, then describe the specific design of our oracle, and show how to integrate information provided by search.

4.1 Motivation and Principle

Each invocation of a state-morphing oracle compares the pool state $t \in \mathcal{P}$ with new morphed states s. Most of these s will not yet be cached, necessitating a run of π. However, that may be expensive. Our **bound-maintenance oracles** instead aim at making maximal use of the information – upper bounds on optimal plan cost – obtained by running π. We move away from oracles treating each invocation as a separate task, to oracles that make use of the information acquired across all invocations. This means that, when given $t \in \mathcal{P}$, our objective is not just to confirm t as a bug, but also to check whether we can retrospectively classify *previous* $t' \in \mathcal{P}$ as bugs and to leverage as much information as possible to help find further bugs in *subsequent* invocations.

Accordingly, state comparisons s vs. t in this new oracle family are not done only to show that t is a bug, but can go in both directions. We use $\mathcal{D}(s,t)$ in order to attempt to prove t to be a bug and $\mathcal{D}(t,s)$ attempting to show s to be a bug. We operationalize this by sharing upper bounds $u(t) \geq h^*(t)$ across oracle invocations. Obviously, for each state t on which π is run, $c^\pi(t)$ is such an upper bound, and whenever we can decrease it further, we have confirmed t to be a bug. The formal basis for this is straightforward:

Proposition 3 (Decreasing Upper Bounds)**.** Let $\mathcal{D}$ be a dominance function, $s,t \in \mathcal{S}$ with $\mathcal{D}(s,t) > -\infty$, and $u_s \in \mathbb{R}^{0+}$ such that $h^*(s) \leq u_s$. Then $h^*(t) \leq u_s - \mathcal{D}(s,t)$.

Proof. Because of $h^*(s) \leq u_s < \infty$, we have that $\mathcal{D}(s,t) \leq h^*(s) - h^*(t)$, and hence $-\infty < h^*(s) - h^*(t)$, so that $h^*(t) < \infty$. Furthermore, $\mathcal{D}(s,t) \leq h^*(s) - h^*(t)$ implies $h^*(t) \leq h^*(s) - \mathcal{D}(s,t)$, and thus $h^*(t) \leq u_s - \mathcal{D}(s,t)$. □

4.2 Oracle Design

Algorithm 3 shows pseudo-code. For each state t, we denote our stored upper bound on $h^*(t)$ by $u(t)$; $u(t)$ is initialized with ∞ for all states. In the BMO procedure, we first run π on the current pool state t, and then compare t to a number of states s already known to the oracle. For simplicity, we abstract from how the states are selected and from the order in which they are considered. Function getCompStates returns all states on which the BMO procedure has previously been invoked; note that $c^\pi(s)$ has already been determined for all these states, incurring no policy-execution overhead.

In the comparison between s and t, we attempt to decrease both $u(s)$ and $u(t)$ by applying Proposition 3. If we are able to decrease $u(s)$ or $u(t)$ below $c^\pi(s)$ or $c^\pi(t)$, we flag the respective state as a bug. Procedures updateAncestors and lookaheadSearch are enhancements discussed below. Setting them aside for the moment, BMO is obviously sound. The key is that $u(s)$ and $u(t)$ are always upper bounds for $h^*(s)$ and $h^*(t)$, respectively. It is easy to see that this invariant is preserved whenever we change $u(s)$ and $u(t)$.

The updateAncestors procedure is simple, so we do not include pseudo-code for it. It does precisely what its name suggests: it traverses the ancestors of its input state (s or t in Algorithm 3) under π, updates their upper bounds, and flags them as bugs whenever that is possible. The only question here is what are "the" ancestors under π. One could, in principle, find all such ancestors by backward search following

Algorithm 3: Bound-maintenance oracle

```
Procedure BMO($t$)
    $u(t) \leftarrow \min(u(t), c^{\pi}(t))$;
    for $s \in$ getCompStates($t$) do
        if $s = t$ then continue;
        if $u(s) < \infty \wedge \mathcal{D}(s,t) > -\infty$ then
            $u(t) \leftarrow \min(u(t), u(s) - \mathcal{D}(s,t))$;
        if $u(t) < \infty \wedge \mathcal{D}(t,s) > -\infty$ then
            $u(s) \leftarrow \min(u(s), u(t) - \mathcal{D}(t,s))$;
            if $u(s) < c^{\pi}(s)$ then
                flagAsBug($s$);
                updateAncestors($s$);
    if $u(t) < c^{\pi}(t)$ then
        flagAsBug($t$);
        updateAncestors($t$);
    lookaheadSearch($t$);    // optional, see Section 4.3
```

only transitions taken by π. Yet, that would be expensive, especially with a costly policy representation like ASNets. Hence, we instead merely remember the parent/child pairs (s, s') from the transitions $s \xrightarrow{\pi(s)} s'$ previously encountered during the testing process. The updateAncestors procedure follows these transitions from child to parent.

Beyond calling Algorithm 3 on the pool states $t \in \mathcal{P}$, we also call it on every state r traversed by a policy run when determining $c^{\pi}(t)$. The reason for this is that, if we succeed in showing r to be a bug, then updateAncestors(r) will flag the pool state t to be a bug as well. Furthermore, the additional bound updates done during these algorithm calls may help to detect additional bugs. The calls on r are done in reverse order, backwards along the policy path, to maximize information propagation. To keep the runtime overhead at bay, we do not run lookaheadSearch in these BMO calls.

Observe that, given these enhancements, states r flagged as bugs by BMO may not be pool states, $r \notin \mathcal{P}$, at the time they are being flagged. The flag is set to remember r as a bug in case it is added to $\mathcal{P}$ later on.[3]

4.3 Integrating Lookahead Search

The oracle as introduced so far relies exclusively on state-comparison information provided by the simulation relation and policy runs. However, as the oracle is based on maintaining upper bounds on h^*, other sources of such upper bounding information can be readily integrated. An obvious such information source is search. We next show how to enhance bound-maintenance oracles with a simple lookahead search procedure; apart from finding actual plans, this also expands the state set considered, and hence the information propagated by state comparisons. In Section 4.4, we will discuss the integration of external search-based oracles.

[3]One could of course report r as a bug anyway, regardless whether or not $r \in \mathcal{P}$ eventually. However, this would lead to many closely correlated bug states, in particular ones traversed by the same policy run. The pool alleviates this problem as its member states are generated randomly. In our experiments, oracle performance is compared on pool states exclusively.

Algorithm 4: Lookahead search for BMO

```
Procedure lookaheadSearch($t$)
    // Conduct GBFS for $n$ steps
    foreach state $s$ visited with $s \neq t$ do
        if $G \subseteq s$ then $u(t) \leftarrow \min(u(t), g(s))$;
        for $v \in$ getCompStates($s$) do
            if $v = s$ then continue;
            if $u(v) < \infty \wedge \mathcal{D}(v,s) > -\infty$ then
                $u(s) \leftarrow \min(u(s), u(v) - \mathcal{D}(v,s))$;
                updateAncestors($s$);
        // next line only if $c^{\pi}(s)$ is cached already
        if $u(s) < c^{\pi}(s)$ then flagAsBug($s$);
        $u(t) \leftarrow \min(u(t), g(s) + u(s))$;
    if $u(t) < c^{\pi}(t)$ then flagAsBug($t$);
    updateAncestors($t$);
```

As specified by the pseudo-code in Algorithm 4, the lookaheadSearch procedure conducts greedy best-first search (GBFS) for a fixed number n of search steps. For every state in the search, we update bounds, perform state comparisons, update ancestors, and flag bugs. Specifically, this procedure is able to infer upper bounds on $h^*(t)$ (other than ∞) if it reaches a goal state; if it reaches a state r for which we already know that $u(r) < \infty$; or if it reaches a state r for which we can infer an upper bound $u(r) < \infty$ by comparing it to other states.

In our implementation, we use h^{FF} (Hoffmann and Nebel 2001) as the heuristic function. We set n to relatively small values, up to 1000; for larger values of n, we empirically found that the overhead typically outweighs the additional information. Intuitively, this is because of all the additional work our lookahead procedure does, beyond just search. But we can also integrate information from purer standard search methods, as described next.

4.4 Combination with External Oracles

We can integrate information provided by any external oracle (or any procedure really), so long as that computes an upper bound on h^* for the input state. As we shall see, for search-based external oracles, this combination yields excellent performance in practice. Intuitively, this is because our oracles employing simulations are designed to be fast, while search-based oracles invest more resources and accordingly identify deeper information. These two complement each other, and the deeper information identified can feed back into, and boost, our bound maintenance.

Algorithm 5 sketches our realization of this idea. We run the bound-maintenance oracle first. If it is not successful, we run the external oracle. For our pseudo-code here, we assume that the external oracle returns the upper bound it computed (even if determining t to be a bug, as in that case the bound may help our oracle to determine yet more bugs).

Algorithm 5: Combined oracle

```
Procedure combinedOracle($t$)
    BMO($t$);
    if $u(t) < c^\pi(t)$ then return;
    $u_t \leftarrow u(t)$;
    $u(t) \leftarrow \min(u(t), \text{externalOracle}(t))$;
    if $u(t) = u_t$ then return;
    for $s \in \text{getCompStates}(t)$ do
        if $s = t$ then continue;
        if $u(t) < \infty \wedge \mathcal{D}(t,s) > -\infty$ then
            $u(s) \leftarrow \min(u(s), u(t) - \mathcal{D}(t,s))$;
            if $u(s) < c^\pi(s)$ then flagAsBug($s$);
    updateAncestors($t$);
```

Crucially, if the external oracle returns a better bound than we had known so far, we can store that information and propagate it into our bound-maintenance oracle, making this combination much more than merely a portfolio of both oracles. We conduct comparisons to classify old states as bugs, update the cost bounds of ancestor states, and use the improved bounds for state comparisons in subsequent invocations of the oracle. The improved bound for the single state (t) on which the external oracle is run can thus propagate into improved bounds for many other states by means of the quantitative simulation relation.

Indeed, this adds an essential new ability to our oracles: if upper bounds come exclusively from policy runs, then, like the metamorphic oracles of Eniser et al. (2022), we can only detect bugs based on comparing policy runs across states. If, however, upper-bound information is added from search, our approach goes strictly beyond metamorphic testing.

5 Experiments

We ran experiments to assess the performance of our oracles – their ability to detect bugs, and the runtime needed for that – compared to other oracles. We next describe the experiment setup, then summarize our results.[4]

5.1 Setup

Our experimental evaluation is firmly based on the work of Steinmetz et al. (2022), i.e., our implementation, benchmarks, and learned policies are based on their framework, which builds on NeuralFD (Ferber, Hoffmann, and Helmert 2020), which in turn builds on FD (Helmert 2006).

We consider ASNet (Toyer et al. 2018, 2020) policies. ASNets are neural networks specifically designed for learning action policies in AI planning. ASNets are trained for a domain, learning a policy that can be instantiated for different instances. Hence, when testing an ASNet policy on multiple tasks of the same domain, we test instantiations of the same policy rather than different policies.

We use the same set of benchmarks and ASNet policies as Steinmetz et al. (2022). On all of these benchmark tasks, the policy solves the initial state (to ensure a certain level of policy quality, as testing very bad policies is not meaningful). We omit the Spanner domain as, there, the ASNet policy performs so well that Steinmetz et al. did not find any bug, and the same was true in our preliminary experiments. Due to technical difficulties (related to our CPU cluster) we also omit the two largest instances of VisitAll.

Regarding the oracles compared, from our own oracles, we run one configuration of the state-morphing oracle (SMO) and several variants of the bound-maintenance oracles (BMO). SMO compares each test state $t \in \mathcal{P}$ to a maximum of 10 morphed states (*maxsize* $= 10$ in Algorithm 2). Larger values of *maxsize* incur too much overhead from running ASNet policies on the morphed states. For BMO, we experiment with multiple limits on the number of expanded states in the lookahead search, $n \in \{0, 10, 100, 1000\}$. We also experiment with a combined oracle, according to Algorithm 5, using BMO together with the search-based Aras/EHC external oracle, which we will describe shortly.

All our SMO and BMO oracles use the same dominance functions. We use the configuration suggested by Torralba (2017), which decomposes the planning task into factors using merge-and-shrink (Helmert et al. 2014) with the DFP strategy (Dräger, Finkbeiner, and Podelski 2009; Sievers, Wehrle, and Helmert 2014) and a size limit of 10 000 transitions per factor. This results in a good trade-off between runtime and informativeness: non-atomic factors improve the latter; larger size limits deteriorate runtime without gaining much or any informativeness.

As competing oracles for comparison, we run:

- **Planner oracles** attempt to find a plan $\vec{a}$ for $t \in \mathcal{P}$, flag t as a bug if $c(\vec{a}) < c^\pi(t)$. Steinmetz et al. (2022) did not run this type of oracle, but we include it here as it constitutes an obvious option. We run A^* with LM-Cut (Helmert and Domshlak 2009); Enforced Hill Climbing (EHC) with h^{FF} (Hoffmann and Nebel 2001); as well as GBFS using h^{FF} with a limit of 1000 search steps, to confirm that the performance of BMOs is not simply due to running this form of GBFS in lookahead search.
- Steinmetz et al.'s (2022) aforementioned **Aras oracle** uses the plan-improvement tool Aras (Nakhost and Müller 2010). As this oracle is only applicable if π finds a plan, we combine it with the EHC oracle: if π solves t then we invoke Aras, else we invoke EHC.
- Steinmetz et al.'s (2022) aforementioned **look+policy oracle** runs depth-first search up to depth 2, and runs π on every leaf state. For fairer comparison to SMO, we apply the same optimization, cutting off the execution of π on leaf state s as soon as $g(s) + c^\pi(s) \geq c^\pi(t)$.

Steinmetz et al. (2022) also run the aforementioned invertibility oracles applicable to the special case of invertible or undoable actions: detecting solvable states not solved by π ("qualitative bugs" in Steinmetz et al.'s terminology) is trivial there, as all states are solvable; states where π's plan is sub-optimal ("quantitative bugs") can be found by going back to a preceding state and using π from there. The latter finds hardly any bugs in Steinmetz et al.'s experiments; the former is trivial and could be used as a pre-process to any

[4]Our code, the benchmarks, and the policies are available at https://gitlab.com/Jan.Eisenhut/metamorphic-test-oracles.

Domain	$\#\Pi$	$\lvert J\rvert$	Oracle recall: Percentage of states in joint test set J flagged as bugs											
			A^*	EHC	Aras/ EHC	look $+ \pi$	GBFS- 1000	SMO	BMO- 0	BMO- 10	BMO- 100	BMO- 1000	BMO-100 + Aras/EHC	Best Of
Blocks	24	481			28.5	**59.7**	0.0	0.0	0.0	6.9	10.2	10.2	34.7	69.2
Floortile	14	555	3.2	0.0	4.9	4.1	0.0	1.1	0.5	0.7	0.7	0.7	**5.0**	5.2
Gripper	35	1966	83.8	64.0	**84.1**	45.6	64.0	11.0	32.9	63.5	84.0	84.0	**84.1**	84.1
MBlocks	6	696	**9.6**	0.9	8.8	5.6	2.9	0.7	0.0	2.0	2.9	4.6	**9.6**	11.6
Satellite	16	740	53.0	29.1	54.5	48.9	43.9	42.7	52.7	53.0	53.1	53.1	**58.1**	58.4
Scanalyzer	50	1801	15.0	2.9	18.6		4.1	6.4	1.6	5.1	6.7	6.7	**19.3**	21.0
Storage	7	1170	**32.6**	26.6	**32.6**	28.9	25.0	10.8	3.8	11.1	23.2	25.5	**32.6**	32.6
Transport	24	2797	34.4	17.4	35.3	29.5	20.1	25.0	27.5	28.5	29.3	29.3	**35.4**	35.6
VisitAll	19	400	76.5	38.0	69.5	52.0	43.2		62.3	63.7	63.7	63.7	**82.0**	85.0

Table 1: Oracle recall (see text). $\#\Pi$: number of tasks in each domain. J: joint pool, processed by all compared oracles, across all tasks. We exclude an oracle from this comparison if its average number of processed states per task is < 25. In the "BestOf" column, we flag all bugs identified by any oracle. Numbers given in oracle names are state expansion limits.

oracle. As our focus is on generally applicable oracles that combine both qualitative and quantitative bug detection, we do not consider invertibility oracles here. For the same reason, we do not consider the perfect qualitative-bug oracles that Steinmetz et al. design for their non-invertible domains.

We execute the two phases of the testing procedure, i.e., the generation of pool states and the invocation of the oracle on these, separately. In the first phase, we collect the pool $\mathcal{P}$, run the policy on each $t \in \mathcal{P}$, and store the results. We attempt to generate 200 pool states, with a time limit of 8 hours on each planning task. To evaluate an oracle, we load the list of pool states $t \in \mathcal{P}$ as well as all action choices the policy took when being run on these pool states. We invoke the oracle on each t, with a time limit of 4 hours for the overall experiment. For all search-based oracles, we use a time limit of 30 minutes per t. For our oracles, the runtime taken for computing the dominance function is counted as part of the oracle evaluation process. All experiments were conducted on a cluster of Intel E5-2660 processors running at 2.20 GHz with a memory limit (per run) of 16 GiB.

5.2 Results

The relevant performance aspects for test oracles are their ability to detect bugs and the required runtime. We consider the former first. We refer to the fraction of pool states identified to be bugs as oracle **recall**. Table 1 shows the data.

As not all oracles process the entire pool $\mathcal{P}$ within the 4-hour time limit, we evaluate recall only on the **joint pool** J, i.e., the subset of $\mathcal{P}$ processed by every oracle. To maximize $|J|$, each oracle processes $\mathcal{P}$ in the same order, and we omit an oracle from a domain if it processes < 25 pool states per task on average.

The results for SMO show that comparing pool states to morphed states with dominance functions – the most direct application of Eniser et al.'s (2022) idea to our setting – is a viable approach for identifying bugs. However, comparing only against 10 states is often insufficient, so SMO generally performs poorly compared to other oracles.

BMOs, on the other hand, are competitive in several domains. The baseline, without any search, is already very effective. Performance strongly increases when using lookahead search in addition to state comparisons. In particular, BMO with a limit of 1000 state expansions consistently outperforms GBFS-1000, which uses the exact same lookahead search but in isolation; so the performance of BMO-1000 is really due to the *combination* of information sources.

The combined oracle (BMO-100 + Aras/EHC) inherits the strengths of both its components, consistently dominating them across all domains. In 5 of the 9 domains – Blocks, MBlocks, Satellite, Scanalyzer, VisitAll – it is strictly better than both its components, again strongly attesting to the synergy between dominance relations and search here. Furthermore, also compared to all other oracles, our combined oracle is dominant, being worse only in Blocksworld, and being strictly better in Satellite, Scanalyzer, and VisitAll. Indeed, the combined oracle is even on par or almost on par (within 0.3 percentage points) with BestOf in 5 of the 9 domains. Overall, the combination of BMO with search is by far the most reliable, and overall most performant, oracle.

Let us now turn to runtime, which is also of crucial importance to users of testing technology, in particular users waiting for bugs to be reported. Figure 1 shows the number of bugs identified by each oracle over time.

As the data shows, BMOs are generally faster than search-based oracles, except for GBFS-1000 with its small search limit. Thus, together with their strong recall (in difference to SMO) their curves rise up earlier than those of the other oracles here. Furthermore, as we are counting all identified bugs $t \in \mathcal{P}$ here, not only the jointly tested states $t \in J$ as in Table 1, the superior speed also enables BMO-1000 to dominate all other oracles (except for the combined oracle) in Gripper, Satellite, and VisitAll. In half of the six domains where BMO-1000 ends up finding clearly fewer bugs than a search-based oracle, that oracle only catches up after a significant amount of time (>30 min in Blocks, Scanalyzer, Transport). BMOs are thus particularly useful if the objective is to find many bugs quickly (which makes sense for debugging as an interactive process).

The combined oracle starts up more slowly than pure BMO, but otherwise it typically dominates all other oracles, often by a substantial margin. The most significant exceptions are look+policy in Blocksworld, and A^* in Stor-

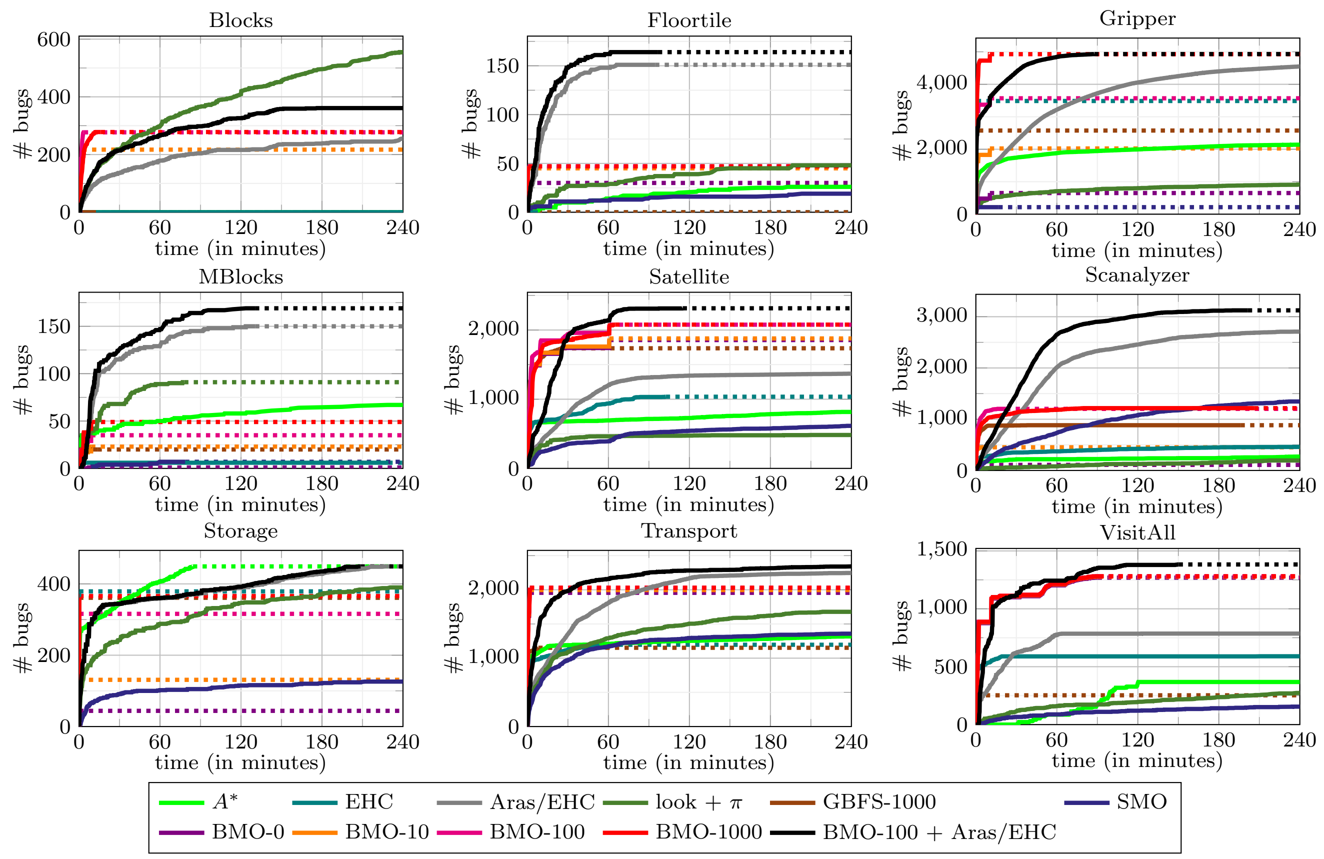

Figure 1: Total number of bugs reported per domain, over time. A solid line turning dashed indicates the time point an oracle has terminated on all domain tasks. We do not include the time required for generating $\mathcal{P}$, which is identical for all oracles.

age. Compared to its components, the combined BMO-100 +Aras/EHC oracle improves both recall and runtime with respect to Aras/EHC on its own; with respect to BMO, the main beneficial effect is recall.

It is worth remarking that some of our design choices and results are influenced by the focus on ASNets, which are very slow to evaluate. For faster policy representations, the relative performance of our two types of metamorphic oracles – SMO and BMO – may change, as SMOs would be able to process more morphed states, increasing their recall. However, relative to competing oracles, no major changes are expected as search is still more costly than dominance-checking, and the synergy between dominance and search information in our combined oracle would still be the same. That said, these are merely informed guesses, and confirming them experimentally remains a topic for future research.

6 Conclusion

Learned action policies are gaining traction for decision making in dynamic environments, and techniques to gain trust in such decisions are becoming increasingly important. Testing is one natural means to do so, but has received little attention in the planning community as yet. Adopting the framework by Steinmetz et al. (2022), and inspired by the ideas of Eniser et al. (2022), here we introduced fully automated test oracles for the test objective of additive-cost minimization in classical planning, leveraging prior work on dominance functions. Our experiments are highly encouraging, showing that bugs can be found much more quickly this way, and showing that the combination with (some) search yields the strongest test oracles for this setting, at this time.

An interesting aspect not yet covered by our work is the “gravity” of bugs, $c^{\pi}(t) - h^{*}(t)$, termed the “testing-objective gap” by Steinmetz et al. (2022). Our oracles provide lower bounds on this gap, and BMO could be tailored to maximize the gap (rather than finding many bugs quickly). Beyond this extension of our current focus, interesting future work directions include the extension to probabilistic planning (necessitating the same extension of dominance functions); as well as the use of bug states for targeted re-training making testing a part of a larger RL loop.

Acknowledgments

This work was funded by the European Union’s Horizon Europe Research and Innovation program under the grant agreement TUPLES No 101070149, as well as by DFG Grant 389792660 as part of TRR 248 (CPEC, https://perspicuous-computing.science).

References

Akazaki, T.; Liu, S.; Yamagata, Y.; Duan, Y.; and Hao, J. 2018. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. In *22nd International Symposium on Formal Methods (FM'18)*, 456–465.

Chen, T. Y.; Cheung, S. C.; and Yiu, S. 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical Report HKUST–CS98–01, HKUST.

Daum, J.; Torralba, Á.; Hoffmann, J.; Haslum, P.; and Weber, I. 2016. Practical Undoability Checking via Contingent Planning. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS'16)*, 106–114.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2009. Directed model checking with distance-preserving abstractions. *International Journal on Software Tools for Technology Transfer*, 11(1): 27–37.

Dreossi, T.; Dang, T.; Donzé, A.; Kapinski, J.; Jin, X.; and Deshmukh, J. V. 2015. Efficient Guiding Strategies for Testing of Temporal Properties of Hybrid Systems. In *7th International Symposium NASA Formal Methods (NFM'15)*, 127–142.

Eniser, H. F.; Gros, T.; Wüstholz, V.; Hoffmann, J.; and Christakis, M. 2022. Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*.

Ernst, G.; Sedwards, S.; Zhang, Z.; and Hasuo, I. 2019. Fast Falsification of Hybrid Systems Using Probabilistically Adaptive Input. In *16th International Conference on Quantitative Evaluation of Systems (QEST'19)*, 165–181.

Ferber, P.; Hoffmann, J.; and Helmert, M. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI'20)*, 2346–2353. IOS Press.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'19)*, 631–636. AAAI Press.

Gentilini, R.; Piazza, C.; and Policriti, A. 2003. From Bisimulation to Simulation: Coarsest Partition Problems. *Journal of Automated Reasoning*, 31(1): 73–103.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS'18)*, 408–416. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge & Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the Association for Computing Machinery*, 61(3): 16:1–16:63.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS'18)*, 422–430. AAAI Press.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI'21)*.

Koren, M.; Alsaif, S.; Lee, R.; and Kochenderfer, M. J. 2018. Adaptive Stress Testing for Autonomous Vehicles. In *IEEE Intelligent Vehicles Symposium (IV'18)*, 1–7. IEEE.

Lee, R.; Mengshoel, O. J.; Saksena, A.; Gardner, R. W.; Genin, D.; Silbermann, J.; Owen, M. P.; and Kochenderfer, M. J. 2020. Adaptive Stress Testing: Finding Likely Failure Events with Reinforcement Learning. *Journal of Artificial Intelligence Research*, 69: 1165–1201.

Milner, R. 1971. An Algebraic Definition of Simulation Between Programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI'71)*, 481–489.

Nakhost, H.; and Müller, M. 2010. Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*, 121–128.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized Label Reduction for Merge-and-Shrink Heuristics. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, 2358–2366.

Stahlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS 2022)*.

Steinmetz, M.; Fišer, D.; Enişer, H. F.; Ferber, P.; Gros, T.; Heim, P.; Höller, D.; Schuler, X.; Wüstholz, V.; Christakis, M.; and Hoffmann, J. 2022. Debugging a Policy: Automatic Action-Policy Testing in AI Planning. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS'22)*.

Torralba, Á. 2017. From Qualitative to Quantitative Dominance Pruning for Optimal Planning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 4426–4432.

Torralba, Á. 2018. Completeness-Preserving Dominance Techniques for Satisficing Planning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, 4844–4851.

Torralba, Á.; and Hoffmann, J. 2015. Simulation-Based Admissible Dominance Pruning. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, 1689–1695.

Toyer, S.; Thiébaux, S.; Trevizan, F. W.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *Journal of Artificial Intelligence Research*, 68: 1–68.

Toyer, S.; Trevizan, F.; Thiebaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*.