Debugging a Policy: Automatic Action-Policy Testing in AI Planning

Marcel Steinmetz,[1] Daniel Fiser,[1] Hasan Ferit Eniser,[2] Patrick Ferber,[1,4] Timo P. Gros,[1] Philippe Heim,[1] Daniel Höller,[1] Xandra Schuler,[1] Valentin Wüstholz,[3] Maria Christakis,[2] Jörg Hoffmann[1,5]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, ⟨lastname⟩@cs.uni-saarland.de
[2] MPI-SWS, Kaiserslautern and Saarbrücken, Germany, {hfeniser, maria}@mpi-sws.org
[3] ConsenSys, Kaiserslautern, Germany, wuestholz@gmail.com
[4] University of Basel, Basel, Switzerland, patrick.ferber@unibas.ch
[5] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany

Abstract

Testing is a promising way to gain trust in neural action policies π. Previous work on policy testing in sequential decision making targeted environment behavior leading to failure conditions. But if the failure is unavoidable given that behavior, then π is not actually to blame. For a situation to qualify as a "bug" in π, there must be an alternative policy π' that does better. We introduce a generic policy testing framework based on that intuition. This raises the *bug confirmation* problem, deciding whether or not a state is a bug. We analyze the use of optimistic and pessimistic bounds for the design of *test oracles* approximating that problem. We contribute an implementation of our framework in classical planning, experimenting with several test oracles and with random-walk methods generating test states biased to poor policy performance and/or state novelty. We evaluate these techniques on policies π learned with ASNets. We find that they are able to effectively identify bugs in these π, and that our random-walk biases improve over uninformed baselines.

1 Introduction

The use of neural networks (NN) to learn action policies π is highly successful in game playing (Mnih et al. 2013; Silver et al. 2018), and is 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). A policy π can take real-time decisions in dynamic environments, simply by evaluating it on the current state to obtain the next action. Yet this approach comes with obvious concerns regarding potential policy "bugs", i.e., undesirable or fatal policy behavior. Testing – trying to find cases where such behavior occurs – is a natural paradigm to address this. Automatic test-case generation can serve to assess the quality of π, and ultimately, through extensive testing, to certify that π can be trusted.

Previous work on testing in sequential decision making controls environment behavior (the state-transition selection in an MDP) and tries to find sequences of environment decisions under which a failure condition ϕ becomes satisfied (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 to blame. So this form of testing makes sense only if failures are avoidable by design.

Here we introduce a generally applicable framework for action-policy testing. Our core intuition is that, for a situation to qualify as a "bug" in π, there must be an alternative policy π' that does better. Focusing on states s (rather than sequences of environment disturbances) as the "situations", we formalize this intuition as sub-optimal performance of π on s relative to a *testing objective*. That objective can be a standard optimization objective like additive-cost minimization, but it can also be a simpler objective whose main purpose is testing, like reaching the goal wherever possible, or avoiding a failure condition ϕ wherever possible.

Obviously, given this definition, the *bug confirmation* problem – deciding whether or not a test state s is a bug – is hard. We need to decide, e.g., whether failure can be avoided when starting from a given state s. Towards scalable approaches to do so, we explore *test oracles*: sufficient criteria that can confirm some (though not all) bugs.

A natural question is whether such oracles can be obtained by drawing on the wealth of known optimistic and pessimistic approximations of planning, in particular the extensive work on admissible heuristic functions (e.g., Haslum and Geffner 2000; Edelkamp 2001; Helmert and Domshlak 2009; Helmert et al. 2014; Pommerening et al. 2015; Davies et al. 2015; Trevizan, Thiébaux, and Haslum 2017; Klösner et al. 2021). Can optimistic and/or pessimistic bounds be employed to show that a given policy behavior is necessarily sub-optimal? Analyzing that question, we determine that, unfortunately, this approach boils down to finding a pessimistic bound (an alternative policy) better than π. In particular, optimistic bounds do *not* per se provide any added value for bug-confirmation oracles.

That said, policy-quality improvement can of course be practical, and there are various special cases in which (some) information about plans can be obtained in different ways. We explore these practical matters in classical *finite-domain representation (FDR)* planning. We consider two testing objectives, namely additive-cost minimization (which we will refer to as *quantitative-FDR*) and reaching the goal wherever possible (*qualitative-FDR*). We use test oracles based on plan-quality improvement (Nakhost and Müller 2010) and special-case oracles that apply when all actions are invertible. To generate test states s, we introduce simple *fuzzing*

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

methods based on random walks. We explore fuzzing biases geared towards poor policy performance, and geared towards novelty (Lipovetzky and Geffner 2012, 2017).

We evaluate our techniques on NN action policies learned with ASNets (Toyer et al. 2018, 2020). We run experiments on a collection of benchmark domains from the International Planning Competition (IPC). We find that our testing methods are effective in identifying bugs in ASNet policies, and that our fuzzing biases improve over uninformed baselines.

2 FDR Planning and Policies in FDR

Our approach is, in principle, agnostic to the planning formalism. Nevertheless, to make things concrete, we focus on FDR classical planning as addressed in our experiments.

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 $\mathcal{D}(v)$. A **state** s is a complete variable assignment; we denote the set of all states with $\mathcal{S}$. $I \in \mathcal{S}$ is the **initial state**. The **goal** G is a partial assignment. $\mathcal{A}$ is a finite set of **actions**, each $a \in \mathcal{A}$ is a pair $\langle \mathsf{pre}(a), \mathsf{eff}(a) \rangle$ of **precondition** and **effect**, both partial assignments. c is a cost function $c : \mathcal{A} \mapsto \mathbb{R}^{0+}$. 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, and similarly $s[\![\vec{a}]\!]$ for an action sequence. $\vec{a}$ is a **plan** for Π if $G \subseteq I[\![\vec{a}]\!]$. We denote the summed-up cost of $\vec{a}$ by $c(\vec{a})$.

A **policy** for an FDR task is a function $\pi : \mathcal{S} \mapsto \mathcal{A}$ 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, e.g., if (like ASNets) the policy generalizes over all instances of a domain.

3 What is a "Bug" in a Policy?

We now introduce the core definitions of our framework. We formalize *bugs* based on *testing objectives* taking the form of *value functions*. We discuss this definition, and we introduce an extended notion of *fuzzing bug* which captures the idea of identifying bugs through comparison to other states.

3.1 Testing Objectives and Policy Bugs

A **testing objective** defines which property of a given policy π we will be testing. This can be any desirable property, including standard optimization objectives, but also simpler objectives like reaching the goal wherever possible, or avoiding a failure condition ϕ wherever possible. As a generic framework within which to express such a testing objective, we use **value functions** V mapping policy/state pairs to numeric "goodness" (policy quality) values.

Following standard notations, we denote by $V^\pi(s)$ the value of applying policy π starting from state s. Better values according to the testing objective may either be smaller ones (minimization) or larger ones (maximization). Regardless of which is the case, we write $V^*(s)$ for the **optimal** (minimal or maximal) value of any policy in s.

Given this, an alternative policy π', that improves over π in terms of the testing objective, exists iff $V^\pi(s) \neq V^*(s)$. Accordingly, we define:

Definition 1 (Bug)**.** A state s is a **bug** in π under testing objective V if $|V^\pi(s) - V^*(s)| > 0$. We refer to $|V^\pi(s) - V^*(s)|$ as the **testing-objective gap**.

3.2 Examples and Scope

In our experiments, we consider two testing objectives in FDR classical planning, which we refer to as **quantitative-FDR** and **qualitative-FDR**. The former is the standard objective to minimize plan cost. We formalize this in terms of the following value function:

Definition 2 (Testing Objective: Quantitative-FDR)**.** Let Π be an FDR task with states $\mathcal{S}$, and π a policy for Π. The **quantitative-FDR** testing objective is defined as:

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

The testing objective is to minimize the value of this function. An alternative policy π' thus is better than π on s – showing that s is a bug in π – iff it generates a cheaper plan for s (or any plan at all, if π doesn't).

In qualitative-FDR, we ignore plan cost, focusing exclusively on whether or not the policy reaches the goal. This can be formalized as a simple maximization objective:

Definition 3 (Testing Objective: Qualitative-FDR)**.** Let Π be an FDR task with states $\mathcal{S}$, and π a policy for Π. The **qualitative-FDR** testing objective is defined as:

$$V^\pi(s) := \begin{cases} 1 & \sigma^\pi(s) \text{ is a plan} \\ 0 & \text{otherwise} \end{cases}$$

A policy "optimal" with respect to this testing objective is one that reaches the goal from every solvable state. An alternative policy π' shows s to be a bug in π under this testing objective iff π' solves s while π does not. This form of testing makes sense to ascertain goal-reaching capability as a basic quality of the policy.

While quantitative and qualitative-FDR are our core focus here, Definition 1 and our theoretical analysis of bug confirmation (Section 4) apply much more generally. Indeed the only assumptions needed are that the policy input is a state (the policy output can be a probability distribution instead of a single action), and that the testing objective can be formulated as a value function. This is the case for every known optimization objective in MDP probabilistic planning, including also binary objectives like satisfaction of a goal/failure probability bound which can be formulated similarly to Definition 3. Further examples are oversubscription planning (V sums up rewards over soft goals) and goal-reaching capability in non-deterministic planning (V measure the fraction of policy runs that reach the goal).

Failure testing against a temporal failure formula ϕ can be captured by this simple testing objective:

$$V^\pi(s) := \begin{cases} 1 & \sigma^\pi(s) \not\models \phi \\ 0 & \text{otherwise} \end{cases}$$

A bug in π here is a state where $V^\pi(s) = 0$, but there exists a policy π' with $V^{\pi'}(s) = 1$: an avoidable failure. Prior work used only the first half, $V^\pi(s) = 0$, of this condition.

While Definition 1 and Section 4 apply as stated across the board, practical concerns may of course differ widely across planning formalisms and testing objectives. The practical exploration of policy testing beyond quantitative-FDR and qualitative-FDR remains a huge topic for future work.

3.3 Discussion

Some words are in order on the expected relation between the testing objective V vs. the planning/learning objective used to construct π. If they are the same, then we test whether π achieves its objective, which is natural. Nevertheless, it will often make sense for the objectives to be different. For example, failure avoidance will certainly not be the only objective a policy (like a car driver) is trained on. But testing it makes perfect sense as this is a "must-have", in difference to other objectives (like travel time minimization) that may be harder to test and/or that a learned policy cannot actually be expected to be optimal on. The same applies to qualitative-FDR as a must-have proxy for quantitative-FDR (does the policy find a plan at all?).

Definition 1 may be counter-intuitive in the sense that s can be a bug even though the action decision $\pi(s)$ itself is optimal. What the definition says is "starting from s, something will go wrong somewhere". An alternative more localized definition of "bug" would be states s where $\pi(s)$ is not part of any optimal policy. Yet this definition would be weak in the sense that the absence of such bugs does not imply that π is optimal with respect to the testing objective: policy decisions must, in general, be coordinated across states.

For example, consider the qualitative-FDR testing objective. As action costs are ignored here, the localized definition suffers from what is known as "0-cost cycles". Consider the state space $g \overset{b}{\leftarrow} s_1 \overset{a}{\leftrightarrow} s_2 \overset{b}{\rightarrow} g$ with goal condition g, and consider π where $\pi(s_1) = a$ and $\pi(s_2) = a$. These action choices each are part of a policy that is optimal accordig to the testing objective. Yet π never reaches the goal.

In general, if a policy π is free of localized bugs, i.e., chooses some optimal action in each state, then the resulting state values satisfy the Bellman equation. But this implies optimality only if every fixed point of the Bellman equation is optimal. The latter is not the case, e.g., for qualitative-FDR, quantitative-FDR with 0-cost actions, maximization of goal probability in MDP planning, and (more generally) generalized stochastic shortest path MDPs (Kolobov et al. 2011). For these reasons, here we consider the stronger definition of bug as per Definition 1. The problem of *fault localization* – identifying specific policy decisions, or combinations of policy decisions, causing harm beneath a bug state s – remains a topic for future work.

3.4 Fuzzing Bugs

In program testing, fuzzing methods modify a given input trying to exhibit a bug. We adopt this idea by modifying a given state s_0 through random walks trying to find a bug-state s.[1] We define a concept of bug specific to this context.

Definition 4 (Fuzzing Bug)**.** A state s is a **fuzzing-bug** relative to s_0 if $|V^\pi(s) - V^*(s)| > |V^\pi(s_0) - V^*(s_0)|$.

Fuzzing is successful if it widens the testing-objective gap. Obviously, this concept is a sub-class of bugs:

Observation 5 (Fuzzing Bugs are Bugs)**.** If s is a fuzzing-bug relative to some s_0, then s is a bug.

In general, the same is almost true vice-versa:

Observation 6 (Bugs are (almost) Fuzzing Bugs)**.** Every bug s with non-minimal testing-objective gap $|V^\pi(s) - V^*(s)|$ is a fuzzing-bug relative to some s_0.

In actual fuzzing algorithms however, there will be restrictions on the reachability of s from s_0, invalidating Observation 6 and making 'fuzzing-bug' a true sub-concept of 'bug'. In any case, the main purpose of Definition 4 here is in our analysis of bug confirmation, which may be facilitated by comparing s to s_0.

4 Bug Confirmation

Bug confirmation is the problem of deciding, given a state s, whether or not s is a bug. This problem subsumes (optimal) planning. Yet we want to apply testing to scenarios where planning is infeasible (which is where learning a policy π makes most sense). Hence we need to identify sufficient criteria – **test oracles** – for bug confirmation: approximations that can confirm some (though not all) bugs.

Given the extensive amount of research on optimistic approximations in planning (lower bounds on cost/upper bounds on reward), a natural idea is to leverage such approximations. We next analyze that possibility, in a generic manner valid across all planning formalisms within our scope. We briefly discuss the tools we consider. Then we discuss bug confirmation, first for individual bug states s, then for fuzzing-bugs where we are comparing s to another state s_0.

As previously stated, our findings are mostly negative. Bug confirmation based on optimistic and pessimistic bounds boils down to policy-quality improvement. In particular, optimistic bounds cannot yield any direct added value. These negative results however pertain to the general case, and to the exclusive use of such bounds. We conclude the section by discussing practical matters, introducing the test oracles we use in our experiments.

4.1 Optimistic/Pessimistic Bounds

To formulate and conduct our theoretical analysis in a manner spanning across different testing objectives, we need a generic "better than" notation, abstracting from minimization vs. maximization. For any value function V, we denote:

Definition 7 (Generic "Better Than" Notation)**.**

$$V(s) \prec V(s') :\text{iff} \begin{cases} V(s) < V(s') & \min \text{objective} \\ V(s) > V(s') & \max \text{objective} \end{cases}$$

[1]The start state s_0 here can be an arbitrary state (recall that we denote the initial state with I), e.g. generated by fuzzing beforehand which we do in FDR planning as described in Section 5.

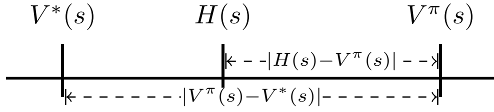

Figure 1: Illustration of Proposition 8, for minimization objectives (smaller is better).

That is, $V(s) \prec V(s')$ iff $V(s)$ is better than $V(s')$. We use $\preceq$, $\succ$, and $\succeq$ accordingly.

The value function approximations we consider are:

- Optimistic bounds h, where $h(s) \preceq V^*(s)$ for all s.
- Pessimistic bounds H, where $V^*(s) \preceq H(s)$ for all s.

For quantitative-FDR, optimistic bounds are the admissible heuristic functions deeply investigated in classical planning (e.g., Haslum and Geffner 2000; Edelkamp 2001; Helmert and Domshlak 2009; Helmert et al. 2014; Pommerening et al. 2015). For qualitative-FDR, optimistic bounds correspond to dead-end detection (setting $h(s) = 0$ if s is recognized as a dead-end, and to 1 otherwise), which has also been deeply addressed (e.g., Hoffmann, Kissmann, and Torralba 2014; Pommerening and Seipp 2016; Steinmetz and Hoffmann 2017). Some optimistic bounding methods beyond classical planning exist (e.g., Domshlak and Mirkis 2015; Trevizan, Thiébaux, and Haslum 2017; Klösner et al. 2021). Pessimistic bounds for all testing objectives can be taken from sub-optimal (satisficing) planning methods.

Our criteria below also use the policy value $V^\pi(s)$. This is trivial to obtain for quantitative-FDR and qualitative-FDR – just execute the policy. In probabilistic settings, $V^\pi(s)$ can be approximated statistically through sample runs. For the purpose of our analysis, we will assume that $V^\pi(s)$ is known, abstracting from the statistical imprecision.

4.2 Bug Confirmation via Bounds

Given that s is a bug iff its testing-objective gap $|V^\pi(s) - V^*(s)|$ is greater than 0, what we need to derive is a lower bound $0 < L \leq |V^\pi(s) - V^*(s)|$ on that gap. It is easy to see that the only possibility for doing so, based on bounds h and H, is the following:

Proposition 8 (Bug Confirmation)**.** If $H(s) \preceq V^\pi(s)$, then $L := |V^\pi(s) - H(s)|$ satisfies $L \leq |V^\pi(s) - V^*(s)|$.

Proof. This simple result holds true because (i) $V^*(s) \preceq H(s)$ by definition of H and (ii) $H(s) \preceq V^\pi(s)$ by prerequisite, so $H(s)$ lies in between the two numbers whose distance we want to lower-bound. Figure 1 illustrates this. □

Importantly, observe that we need both (i) and (ii). For (i), if our approximation of $V^*(s)$ is not guaranteed to be pessimistic, i.e., if $H(s) \prec V^*(s)$ is possible, then the distance $L = |V^\pi(s) - H(s)|$ may be arbitrarily larger than the testing-objective gap $|V^\pi(s) - V^*(s)|$. Note that, in particular, we cannot use an optimistic approximation $h(s)$. Similarly for (ii), if our pessimistic approximation is worse than the policy, i.e., $V^\pi(s) \prec H(s)$, then L may be arbitrarily larger than the testing-objective gap.

In short, the only possibility to use optimistic and pessimistic value-function approximations for bug confirmation is to find a better solution than π. While this may not be surprising (at least not with the benefit of hindsight), we now turn to the question whether it may help to consider s not in isolation, but in comparison to another state s_0. Unfortunately – and perhaps more surprisingly – the answer is "no".

4.3 Fuzzing-Bug Confirmation via Bounds

To confirm s as a fuzzing-bug relative to another state s_0, what we need to show is that the value gap widens from s_0 to s. As the first step in our analysis, we systematize the possible cases. We do so by distinguishing the relation between the size of the policy-value change from s_0 to s vs. the optimal-value change from s_0 to s:

Proposition 9 (Fuzzing-Bug Characterization)**.** s is a fuzzing bug relative to s_0 if and only if one of the following conditions holds:

(a) $|V^\pi(s) - V^\pi(s_0)| > |V^*(s) - V^*(s_0)|$ and $V^\pi(s) \succeq V^\pi(s_0)$
(b) $|V^\pi(s) - V^\pi(s_0)| < |V^*(s) - V^*(s_0)|$ and $V^*(s) \preceq V^*(s_0)$
(c) $|V^\pi(s) - V^\pi(s_0)| = |V^*(s) - V^*(s_0)| \neq 0$ and $V^\pi(s) \succeq V^\pi(s_0)$ and $V^*(s) \preceq V^*(s_0)$

Proof. In case (a), V^π changes more than V^*, and V^π can only become worse, so V^π must do worse at s than at s_0. In case (b), V^π changes less than V^*, and V^* can only become better, to the same conclusion. In case (c), the changes are identical but in different directions.

Clearly, one of (a) – (c) must hold, and in each case the prerequisites on change directions are required, so the case distinction is exhaustive as claimed. □

How to obtain sufficient criteria for Proposition 9? Clearly, (c) is fruitless as we would need to know the exact change in V^* from s_0 to s, which will hardly be possible without knowing $V^*(s_0)$ and $V^*(s)$ in the first place. Cases (a) and (b) can be approximated as follows.

For (a), given our assumption that V^π can be evaluated with sufficient precision, the term we need to approximate is $|V^*(s) - V^*(s_0)|$. Specifically, we need our optimistic and pessimistic approximations of V^* to provide an upper bound $U(s, s_0) \geq |V^*(s) - V^*(s_0)|$ on the optimal value change. This can indeed be obtained, as follows:

Proposition 10 (Fuzzing-Bug Confirmation (a))**.** Let $\delta_1 := |h(s) - H(s_0)|$, $\delta_2 := |H(s) - h(s_0)|$, and $U(s, s_0) := \max(\delta_1, \delta_2)$. Then s is a fuzzing-bug relative to s_0 if $|V^\pi(s) - V^\pi(s_0)| > U(s, s_0)$ and $V^\pi(s) \succeq V^\pi(s_0)$.

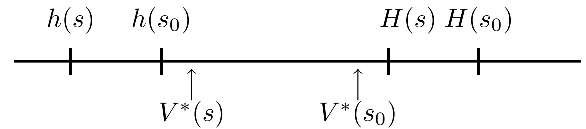

Figure 2: Proposition 10, case $V^*(s) \preceq V^*(s_0)$.

Proof. To see that this holds, observe that, if $V^*(s) \preceq V^*(s_0)$ (see illustration in Figure 2), then $\delta_1 = |h(s) - H(s_0)| \geq |V^*(s) - V^*(s_0)|$ because $h(s)$ lies "to the left of" (is better than) $V^*(s)$ while $H(s_0)$ lies "to the

right of" (is worse than) $V^*(s_0)$. If $V^*(s) \succeq V^*(s_0)$, then $\delta_2 = |H(s) - h(s_0)| \geq |V^*(s) - V^*(s_0)|$ for symmetric reasons. Hence $U(s, s_0) \geq |V^*(s) - V^*(s_0)|$, from which the claim follows directly with Proposition 9 (a). □

Observe that an upper bound on $|V^\pi(s) - V^\pi(s_0)|$ cannot be derived from h and H in any other way. For any other pair $\{l, u\} \subseteq \{h(s), h(s_0), H(s), H(s_0)\}$ other than the ones used in δ_1 and δ_2, one can easily construct scenarios where $|u-l| < |V^\pi(s) - V^\pi(s_0)|$, for both cases $V^*(s) \preceq V^*(s_0)$ and $V^*(s) \succeq V^*(s_0)$. For example, as can be easily seen in Figure 2, $|h(s) - h(s_0)|$ and $|H(s) - H(s_0)|$ do not work.

Let us now consider case (b) of Proposition 9, $|V^\pi(s) - V^\pi(s_0)| < |V^*(s) - V^*(s_0)|$ and $V^*(s) \preceq V^*(s_0)$. Assuming again that we can evaluate V^π, what we need is a lower bound $0 < L(s, s_0) \leq |V^*(s) - V^*(s_0)|$ on the optimal value change, as well as a proof that $V^*(s) \preceq V^*(s_0)$.

Consider the two intervals $I(s) := [h(s), H(s)]$ and $I(s_0) := [h(s_0), H(s_0)]$ limiting the possible $V^*(s)$ and $V^*(s_0)$ values (illustrated in Figure 3. If $I(s) \cap I(s_0) \neq \emptyset$, then $V^*(s) = V^*(s_0)$ is possible, so we cannot handle that case. If $I(s) \cap I(s_0) = \emptyset$, then we can do the following:

Proposition 11 (Fuzzing-Bug Confirmation (b))**.** Say that $I(s_0) \cap I(s) = \emptyset$, and let $L(s, s_0) := |H(s) - h(s_0)|$. Then s is a fuzzing bug relative to s_0 if $|V^\pi(s) - V^\pi(s_0)| < L(s, s_0)$ and $H(s) \prec h(s_0)$.

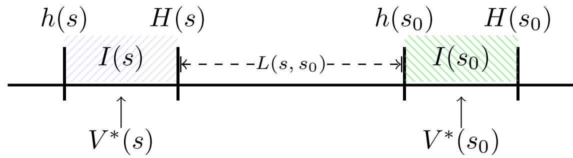

Figure 3: Illustration of Proposition 11.

Proof. With $H(s) \prec h(s_0)$, we get $V^*(s) \prec V^*(s_0)$. With $I(s_0) \cap I(s) = \emptyset$ and $H(s) \prec h(s_0)$, $L(s, s_0) = |H(s) - h(s_0)|$ is exactly the minimum distance between the two intervals, $|H(s) - h(s_0)| = \min\{|v - v'| \mid v \in I(s_0), v' \in I(s)\}$ (see Figure 3). Hence $L(s, s_0) \leq |V^*(s) - V^*(s_0)|$. The claim follows from Proposition 9 (b). □

Observe again that this is the *only* possible approximation based on bounds h and H: our discussion of $I(s)$ and $I(s_0)$ handles both possible cases, so there is nothing else we can do to obtain a lower bound on $|V^*(s) - V^*(s_0)|$. Therefore, Proposition 10 and Proposition 11 together exhaustively cover the space of possibilities for exploiting such bounds. As previously hinted, unfortunately this space of possibilities is dominated by Proposition 8:

Theorem 12. Let s_0 and s be arbitrary states. Consider the same bounds $h \preceq V^*$ and $H \succeq V^*$ in all methods. If Proposition 10 or Proposition 11 confirm s to be a bug, then so does Proposition 8.

Proof. We show that Proposition 10 and Proposition 11 each imply $H(s) \prec V^\pi(s)$. Therefore, if they are satisfied, then s also satisfies Proposition 8 for the considered pessimistic bound. Assume for contradiction that $H(s) \succeq V^\pi(s)$.

Consider first Proposition 10. By assumption, $H(s) \succeq V^\pi(s)$; the second condition of Proposition 10 gives $V^\pi(s) \succeq V^\pi(s_0)$; since $U(s, s)$ is non-negative by definition, the first condition entails that $|V^\pi(s) - V^\pi(s_0)| > 0$, so the inequality in our previous observation is actually strict ($V^\pi(s) \succ V^\pi(s_0)$); and finally $V^\pi(s_0) \succeq h(s_0)$ since h optimistically bounds V^*. We arrive at the following chain of relations: $H(s) \succeq V^\pi(s) \succ V^\pi(s_0) \succeq h(s_0)$. Therefore, $|V^\pi(s) - V^\pi(s_0)| \leq |H(s) - h(s_0)|$. Since $|H(s) - h(s_0)|$ is exactly δ_1 in the claim of Proposition 10, and $\delta_1 \leq U(s, s)$, we conclude that $|V^\pi(s) - V^\pi(s_0)| \leq U(s, s)$. This contradicts the condition of Proposition 10.

If Proposition 11 is satisfied, its prerequisite connects $V^\pi(s_0)$ and $V^\pi(s)$ through the following chain of relations: $V^\pi(s_0) \succeq h(s_0)$ because h is optimistic; $h(s_0) \succ H(s)$ per prerequisite of Proposition 11; $H(s) \succeq V^\pi(s)$ per our assumption. Overall, we get $V^\pi(s_0) \succeq h(s_0) \succ H(s) \succeq V^\pi(s)$ and hence $|V^\pi(s) - V^\pi(s_0)| \geq |H(s) - h(s_0)|$. Now, as argued in the proof of Proposition 11, the minimum distance $L(s, s)$ between the intervals $I(s_0)$ and $I(s)$ is exactly $|H(s) - h(s_0)|$. Put together, we get $|V^\pi(s) - V^\pi(s_0)| \geq L(s, s)$, which is a contradiction to the prerequisite of Proposition 11. □

4.4 Discussion and Practical Special Cases

While it is disappointing that the use of optimistic and pessimistic bounds boils down to finding a pessimistic bound better than $V^\pi(s)$, of course that can be a practical method. Plan-quality improvement has been investigated in some depth in classical and probabilistic planning already (e.g., Tesauro and Galperin 1996; Bäckström 1998; Do and Kambhampati 2003; Chang, Givan, and Chong 2004; Nakhost and Müller 2010; Siddiqui and Haslum 2015). Anytime planning methods can be used for this purpose too. Another interesting option are branch-and-bound and bounded-suboptimal searches, pruning against $V^\pi(s)$.

In our implementation and experiments, we address FDR planning, with the quantitative-FDR and qualitative-FDR testing objectives. We explore two plan-quality improvement methods in this context. First, we leverage the Aras tool (Nakhost and Müller 2010) which performs well and is comparatively easy to connect to. If Aras manages to improve the plan returned by a policy (i.e., the policy run $\sigma^\pi(s)$) then s is a quantitative-FDR bug in π. We refer to this method as the **Aras oracle**. Second, we implement a simple new plan-quality improvement method leveraging the fact that we have a policy π at hand. We run a depth-first lookahead search up to depth d, and we run π on each leaf state t of that search. If an alternate plan (path to t and run of π on t) is cheaper than $\sigma^\pi(s)$, then s is a quantitative-FDR bug in π. We refer to this method as the **Lookahead oracle**. We also experiment with a qualitative-FDR version of this oracle, detecting a bug if π does not reach the goal from s, but does reach the goal from some leaf state t.

Beyond plan-quality improvement, the second major possibility is to consider special cases of planning, and/or information about plans other than optimistic and pessimistic bounds. This can potentially be done in many ways, and remains a large topic for future research. For example, sanity

Algorithm 1: Fuzzer for FDR planning.

```
1  P := {I};
2  while |P| < N and runtime ≤ T do
3      s_0 := uniformRandom(P);
4      l := uniformRandom({1 . . . L});
5      s := s_0;
6      for i = 1 . . . l do
7          A := A[s] \ {a | h^max(s⟦a⟧) = ∞};
8          if PolQualBias then
9              a := policyQualityBias(A);
10         else
11             a := uniformRandom(A);
12         s := s⟦a⟧;
13     if NoveltyFilter then
14         if novelty(s, P) ≤ D then
15             P := P ∪ {s};
16     else
17         P := P ∪ {s};
18 return P;
```

testing may consider only states for which V^* is known; or it may be possible to adapt ideas developed in software testing for scenarios where the correct program output is not known.

In our experiments, we explore two methods applicable to the special case where all actions are "at least undoable" (Daum et al. 2016), i.e., we can repair any negative effects an action may have. In this case, a plan exists for every state so that policy failures are bugs and qualitative-FDR bug confirmation trivializes. We refer to this as the **UndoQual oracle**. For quantitative-FDR bugs, we leverage the action sequence $\vec{a}$ leading from s_0 to s. In the special case of undoability where each action has a corresponding inverse action, one possible plan for s is to go back to s_0 and use π from there. Hence, if $V^\pi(s) - V^\pi(s_0) > c(\vec{a})$, then we know s is a bug. We refer to this as the **InvQuant oracle**.

5 Fuzzing Methods

In our implementation of action-policy testing for FDR planning, we employ fuzzing methods to automatically generate test states s. Algorithm 1 shows the pseudo-code. Recall here that $A[s]$ denotes the set of actions applicable to s, and $s[\![a]\!]$ denotes the state resulting from applying a to s.

The algorithm structure is straightforward: we iteratively build up a test-state pool P of size N by random walks of maximal length L, starting from a state s_0 already in the pool. We choose actions either with a policy quality bias or uniformly, ignoring actions whose outcome state is easily shown to be unsolvable and hence not of interest to policy testing. We add the new state s either if it passes a novelty filter or without any filter. The time limit T is used to cover cases where reaching $|P| = N$ would take too long (or is actually impossible given the novelty filter).

The policy quality bias, switched on by setting the Boolean flag PolQualBias, assigns each action $a \in A$ a weight $weight(a)$, and obtains a probability distribution by normalizing these weights to sum up to 1. We define $weight(a)$ as the length of $\sigma^\pi(s)$ if that reaches the goal. If $\sigma^\pi(s)$ does not reach the goal, we pass $s[\![a]\!]$ to pool insertion (i.e., lines 13 – 17), but set $weight(a) := 0$ to avoid contiguous bug-state regions (which are less interesting than diverse bugs), and to avoid contiguous dead-end state regions in domains where dead-ends exist.

The optional novelty filter, switched on by setting the Boolean flag NoveltyFilter, biases the pool towards states with a high degree of novelty in the sense of Lipovetzky and Geffner (2012; 2017). Namely, we define $novelty(s, P)$ as the size of the smallest tuple t of variable values true in s that is not true in any $s' \in P$. If $D = 1$, this means that every new state s in the pool must contain a variable value not added before; if $D = 2$ then s must contain a new variable-value pair; and so forth. The test pool is thus forced to be more diverse, covering different aspects of the state space.

6 Experiments

Our techniques are implemented on top of NeuralFD (Ferber, Hoffmann, and Helmert 2020), which itself is an extension of FD (Helmert 2006). For the Aras oracle, we connect to the plan-quality improver Aras (Nakhost and Müller 2010) by a sub-process (Aras cannot be easily merged into NeuralFD as the underlying FD versions are very different). The action policies we evaluate our techniques on are learned with ASNets (Toyer et al. 2018, 2020) (see details in Section 6.1 below). We connect the original ASNets source code to NeuralFD using a Python process within C++; the overhead incurred by that connection is negligible.[2]

Our experiments compare five oracles, of which three are for quantitative-FDR (Aras, Lookahead, InvQuant) and two for qualitative-FDR (Lookahead, UndoQual). These oracles have complementary strengths in theory, and our evaluation shows how they compare in practice. We furthermore compare the four fuzzer variants resulting from the two options PolQualBias and NoveltyFilter. Each option constitutes an advanced method (switch on), for which our results show improvements over the respective baseline (switch off).

ASNet policies are large, and are slow to evaluate: typically $\geq$ 0.2s for each call on our benchmarks, which becomes significant as policy runs often have 100s of steps. We hence fix the search depth d in the Lookahead oracle to 2 (avoiding too many leaf states to run ASNets on). For fuzzing, we set the time limit T to 5 hours and the maximum pool size to $N = 200$. We fix the maximum walk length L to 5, which worked well across benchmarks. We fix the novelty filtering level D to 2, as $D = 1$ was too radical in many cases, while $D > 2$ is too expensive. For running the oracle on the pool, we set a time limit of 10 hours (across all pool states). Each configuration is executed with a memory limit of 16 GB on a cluster with AMD EPYC 7543 processors.

6.1 Benchmarks and ASNet Policies

The "benchmarks" required for our research encompass not only planning domains and instances, but also policies for

[2]Our code, the benchmarks, and the ASNet policies are available at https://doi.org/10.5281/zenodo.6323289

those. Here we contribute an initial benchmark collection, that may serve towards a joint benchmark basis for policy testing and other analysis methods (verification, explanation, ...) in the planning community.

We focus on ASNets as a recent and competitive method to learn action policies in PDDL planning benchmarks.[3] As testing is of interest primarily for policies π that have a certain level of quality, for the evaluation of our testing algorithms we exclusively consider benchmark instances solved by π (executing π in the initial state reaches the goal); and we focus on domains with high **policy coverage**, i.e., where π solves a large fraction of instances. Our collection includes (a) policies trained by the original authors of ASNets, as well as (b) additional policies we trained ourselves.

Regarding (a), Toyer et al. (2020) train ASNet policies for three classical-planning domains, namely Blocksworld, MatchingBlocksworld, and Goldminer. Their policies for Goldminer perform very badly (0 test-set coverage), so we ignore that domain. For Blocksworld and MatchingBlocksworld, we included into our benchmark set the exact policies trained by Toyer et al.

For (b), we ran ASNet training on a broad range of IPC domains, using the original ASNets machinery and hyperparameters. We selected the domains with best policy coverage. As two of our oracles are specialized to undoable actions, we first experimented on such domains, for which we obtained high-coverage policies in Gripper, Satellite, Scanalyzer, Transport, and Visitall. We include Floortile as a domain that contains dead-ends, thus contributing this kind of structure to the benchmark basis (along with MatchingBlocksworld). We finally include Spanner from the IPC learning track, which also contains dead-ends, but where learning a simple trick suffices to avoid those. This list of domains is presumably not exhaustive (one can presumably learn high-coverage ASNet policies in more IPC domains), but provides a solid basis for our evaluation purposes.

6.2 Results

Table 1 shows our evaluation of testing results as a function of oracle and fuzzer configuration.

Consider first part (A) of the table, which gives statistics for the benchmark domains. The ASNet policies have perfect policy coverage in Gripper, Scanalyzer, and Spannner; they are weak but non-trivial in MatchingBlocksworld and Storage; they achieve substantial coverage in the other domains. Remember that, for our evaluation in what follows, we consider only the solved benchmark instances.

Part (B) of Table 1 gives statistics about the pools P of test states returned by the fuzzer. For our purposes here, what matters is that the pools are adequate to evaluate qualitative-FDR and quantitative-FDR oracles. The former apply only to $s \in P$ not solved by π (where $\sigma^\pi(s)$ is not a plan), where they try to show that a plan exists; the latter apply only to $s \in P$ solved by π, where they try to show that $\sigma^\pi(s)$ is suboptimal. As Table 1 shows, the number of unsolved $s \in P$ is sufficient everywhere, except in Gripper where all pool states are solved. The number of solved $s \in P$ is small in MatchingBlocksworld, Floortile, Spanner, and VisitAll. Setting PolQualBias=OFF, this improves. The resulting evaluation data (parts (C) – (E) of Table 1) are similar, confirming the observations stated what follows.

Part (C) of Table 1 evaluates the oracles by their **recall**, the fraction of unsolved (solved) pool states $s \in P$ identified to be qualitative-FDR (quantitative-FDR) bugs. For the qualitative-FDR oracles, note first that, on those domains where the UndoQual oracle is applicable, it is actually perfect – all states are solvable so s is a bug iff π does not reach the goal. For each of the three other domains in our collection, we implemented a domain-specific perfect oracle (deciding whether or not a state is solvable), allowing us to include perfect-oracle data throughout. The Lookahead oracle falls behind the perfect oracle, although the difference is often small. No qualitative-FDR bugs are found in Gripper and Spanner (in Gripper, all $s \in P$ are solved so recall among unsolved states is not applicable). Indeed, the ASNet policy seems to be able to solve all states in these domains.

Among the quantitative-FDR oracles, InvQuant finds only few bugs (it's rarely better to go back to s_0 and use π from there). The other two oracles both make a larger computational investment (using search), which pays off by finding more bugs. Aras often works better, but Lookahead has a large advantage in Blocksworld and Satellite, and a small one in Scanalyzer. Computing a perfect oracle (optimal planning) is completely infeasible on most of these benchmarks. There cannot be any quantitative-FDR bugs in Spanner as all plans in this domain are optimal.

Part (D) of Table 1 evaluates the impact of the policy quality bias in fuzzing, varying the value of the PolQualBias switch while using a best-of oracle (perfect for qualitative-FDR, union of all oracles for quantitative-FDR). For qualitative-FDR, we show recall among all $s \in P$, rather than only among the unsolved $s \in P$, because the latter would simply be 100.0 in most domains. Intuitively, recall in P using the perfect oracle measures the frequency with which the fuzzer generates qualitative-FDR bug states. For quantitative-FDR, we stick to the previous evaluation of recall among solved $s \in P$, measuring the frequency with which solved states generated by the fuzzer are recognized by our oracles to be quantitative-FDR bugs. As the data shows, the policy quality bias tends to improve both frequencies, often dramatically, and with only minor exceptions.

Consider finally part (E) of Table 1, which evaluates the impact of the novelty filter (using the same oracles and recall definitions as in (D)). As the data shows, recall is typically not much affected by the switch. So, as far as the identification of bugs is concerned, the random walks as a baseline have sufficient exploration. The advantage of novelty filtering lies in the diversity of the bugs identified, which is beneficial in terms of broad testing. The fraction of bug states that contain a unique fact pair is a very direct measure of this effect, which as expected tends to go up with novelty

[3] ASNets output a probability distribution P over all actions. We obtain our deterministic policy decisions $\pi(s)$ from this by selecting the highest-probability applicable action, $\arg\max_{a \in A[s]} P(a)$. Furthermore, we exclusively consider ASNets whose input is the state, not Toyer et al.'s variants taking also heuristic values and/or action history (such extensions of our framework are future work).

(A) Benchmarks			(B) Pool P		(C) Oracles Comparison						(D) PolQualBias				(E) NoveltyFilter											
					recall (%)						recall (%)				recall (%)				bugs w/ unique fact pair (%)				#bug regions			
					qual ($P \setminus P_\pi$)			quant (P_π)			qual (P)		quant (P_π)		qual (P)		quant (P_π)		qual		quant		qual		quant	
Domain	Σ	Σ_π	$\|P\|$	$\|P_\pi\|$	undo	look	perf	inv	look	aras	off	on	off	on	off	on	off	on	off	on	off	on	off	on	off	on
Toyer et al. (2020) ASNet Policies																										
Blocksworld	30	24	144.6	28.3	100.0	65.3	100.0	5.0	49.0	20.1	8.5	64.4	33.8	56.5	64.3	64.4	61.6	56.5	64.2	70.9	96.2	96.4	4.4	7.6	3.4	4.2
MatchingBlocks	51	6	139.3	8.0	–	0.5	1.1	–	8.3	60.0	5.9	1.1	53.5	60.0	4.8	1.1	50.0	60.0	89.4	100.0	100.0	100.0	0.7	0.5	0.5	1.0
Own ASNet Policies on IPC Benchmarks																										
Floortile	20	14	186.2	1.5	–	0.7	4.6	–	33.7	49.0	4.1	4.6	37.9	49.0	6.1	4.6	50.0	49.0	90.6	91.9	97.6	97.6	0.4	0.4	0.6	0.6
Gripper	35	35	58.3	58.3				0.5	15.2	90.1	0.0	0.0	87.1	90.2	0.0	0.0	87.8	90.2			66.9	78.3	0.0	0.0	18.4	29.2
Satellite	20	16	114.0	9.3	100.0	88.2	100.0	1.0	70.5	37.4	16.9	85.7	38.3	74.9	88.7	85.7	70.0	74.9	65.9	72.9	85.3	97.7	4.4	5.2	0.9	2.4
Scanalyzer	50	50	63.9	19.2	100.0	40.7	100.0	0.1	12.3	7.9	4.0	28.5	13.8	16.3	29.0	28.5	15.3	16.3	64.9	73.0	99.0	100.0	3.3	5.8	1.9	2.0
Spanner	40	40	142.8	2.9	–	0.0	0.0	–	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0					0.0	0.0	0.0	0.0
Storage	30	7	145.3	58.4	100.0	88.5	100.0	0.4	13.9	23.4	21.0	44.5	17.1	23.4	51.3	44.5	24.7	23.4	38.8	58.3	98.8	98.3	1.1	7.1	4.9	5.6
Transport	60	24	151.2	51.4	100.0	97.8	100.0	1.5	37.1	48.6	17.7	50.9	46.7	48.8	56.2	50.9	46.4	48.8	37.0	58.4	78.7	92.8	1.1	5.7	8.7	14.8
VisitAll	79	21	78.7	3.2	100.0	19.8	100.0	2.9	12.2	15.4	79.9	86.9	28.8	23.1	87.1	86.9	26.2	23.1	71.2	81.9	100.0	100.0	6.3	9.5	0.5	0.7

Table 1: Average statistics per domain. (A) total number of instances (Σ), number of instances solved by policy (Σ_π). (B) average pool size ($|P|$), average number of pool states solved by policy ($|P_\pi|$). (C) evaluates the oracles, (D) evaluates the PolQualBias switch, (E) evaluates the NoveltyFilter switch. Abbreviations: qualitative-FDR (qual), quantitative-FDR (quant), UndoQual (undo), Lookahead (look), InvQuant (inv), perfect (perf). As indicated, recall for quant is over P_π; for qual, it is over $P \setminus P_\pi$ in (C), and over P in (D) and (E) (see text). Oracles in (D) and (E) are best-of, i.e., perfect for qual, and union of oracles for quant (detecting a bug if any of the oracles does). Fuzzer default setting is PolQualBias=ON and NoveltyFilter=ON. Table entries dashed out "–" mean the method is not applicable, table entries left empty are undefined fractions (division by zero).

filtering. To provide a more semantic perspective, we furthermore measure the number of bug regions, where a "region" is a connected state-space sub-graph of bug states. As the data shows, the number of distinct bug regions typically goes up when switching novelty filtering on.

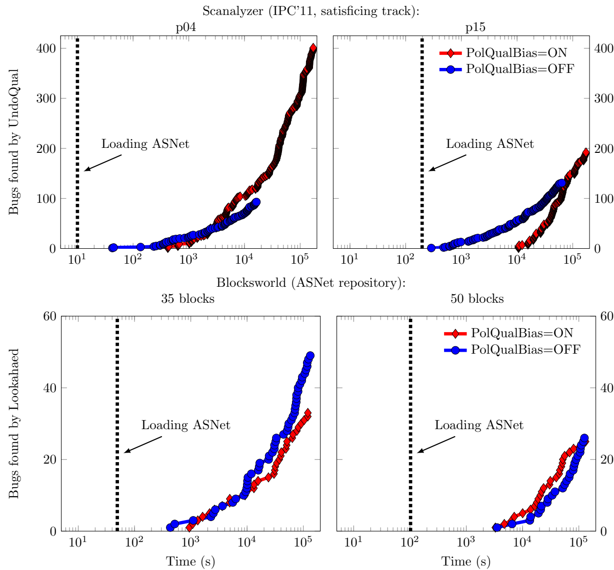

Figure 4: Number of bugs (top: qualitative-FDR with UndoQual, bottom: quantitative-FDR with Lookahead) as a function of time for selected instances. NoveltyFilter = ON, time limit 48h.

To give a view on the debugging process as a function of time spent, Figure 4 shows in-depth data. We consider Blocksworld and Scanalyzer here, where plans are non-trivial and still the ASNet policies have (almost) perfect coverage. We interleave fuzzing with oracle evaluation here, calling the oracle each time a new state is added to the pool.

As the data shows, the runtime overhead of the policy quality bias means that it takes a longer time until the first bugs are found. But given enough time, the biased method tends to work better thanks to its higher bug frequency. The time investment for debugging is substantial – it takes hours rather than seconds to obtain a meaningful result – but seems adequate given high user interest in policy certification. Also, for policies faster to evaluate than ASNets, the effort for debugging will reduce accordingly.

7 Conclusion

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 so far been largely neglected in the planning community. Here we provide a general framework, an analysis of test-oracle design via bounding techniques, and an implementation and empirical evaluation yielding insights into algorithm variant behavior.

We view this as a first step towards an important sub-area of planning research. Our work lays the basis for deeper investigations of oracle design (e.g., drawing on methodologies from software testing), fuzzing methods (e.g., exploring biases based on heuristic functions), fault localization (e.g., trying to identify sub-optimal actions in scenarios without 0-cost cycles), and ultimately targeted re-training making testing a part of a larger RL loop.

Acknowledgments

This work was funded by DFG Grant 389792660 as part of TRR 248 (CPEC, https://perspicuous-computing.science). We thank the anonymous reviewers for their comments, which helped improve this paper.

References

Akazaki, T.; Liu, S.; Yamagata, Y.; Duan, Y.; and Hao, J. 2018. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. In *Intl Symp Formal Methods*.

Bäckström, C. 1998. Computational Aspects of Reordering Plans. *JAIR*, 9: 99–137.

Chang, H. S.; Givan, R.; and Chong, E. K. P. 2004. Parallel Rollout for Online Solution of Partially Observable Markov Decision Processes. *Discrete Event Dynamic Systems*, 14(3): 309–341.

Daum, J.; Torralba, Á.; Hoffmann, J.; Haslum, P.; and Weber, I. 2016. Practical Undoability Checking via Contingent Planning. In *ICAPS*.

Davies, T. O.; Pearce, A. R.; Stuckey, P. J.; and Lipovetzky, N. 2015. Sequencing Operator Counts. In *ICAPS*.

Do, M. B.; and Kambhampati, S. 2003. Improving the Temporal Flexibility of Position Constrained Metric Temporal Plans. In *ICAPS*.

Domshlak, C.; and Mirkis, V. 2015. Deterministic Oversubscription Planning as Heuristic Search: Abstractions and Reformulations. *JAIR*, 52: 97–169.

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 *NASA Formal Methods (NFM)*.

Edelkamp, S. 2001. Planning with Pattern Databases. In *ECP*.

Ernst, G.; Sedwards, S.; Zhang, Z.; and Hasuo, I. 2019. Fast Falsification of Hybrid Systems Using Probabilistically Adaptive Input. In *QEST*.

Ferber, P.; Hoffmann, J.; and Helmert, M. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *ECAI*.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *ICAPS*, 631–636.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *ICAPS*.

Haslum, P.; and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *AIPS*.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *ICAPS*.

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

Hoffmann, J.; Kissmann, P.; and Torralba, Á. 2014. "Distance"? Who Cares? Tailoring Merge-and-Shrink Heuristics to Detect Unsolvability. In *ECAI*.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *ICAPS*.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *AAAI*.

Klösner, T.; Torralba, Á.; Steinmetz, M.; and Hoffmann, J. 2021. Pattern Databases for Goal-Probability Maximization in Probabilistic Planning. In *ICAPS*.

Kolobov, A.; Mausam; Weld, D. S.; and Geffner, H. 2011. Heuristic Search for Generalized Stochastic Shortest Path MDPs. In *ICAPS*.

Koren, M.; Alsaif, S.; Lee, R.; and Kochenderfer, M. J. 2018. Adaptive Stress Testing for Autonomous Vehicles. In *IEEE Intelligent Vehicles Symposium*.

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. *JAIR*, 69: 1165–1201.

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In *ECAI*.

Lipovetzky, N.; and Geffner, H. 2017. A Polynomial Planning Algorithm that Beats LAMA and FF. In *ICAPS*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*.

Nakhost, H.; and Müller, M. 2010. Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement. In *ICAPS*.

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From Non-Negative to General Operator Cost Partitioning. In *AAAI*.

Pommerening, F.; and Seipp, J. 2016. Fast Downward Dead-End Pattern Database. In *UIPC 2016 planner abstracts*, 2–2.

Siddiqui, F. H.; and Haslum, P. 2015. Continuing Plan Quality Optimisation. *JAIR*, 54: 369–435.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Steinmetz, M.; and Hoffmann, J. 2017. State Space Search Nogood Learning: Online Refinement of Critical-Path Dead-End Detectors in Planning. *AI*, 245: 1–37.

Tesauro, G.; and Galperin, G. R. 1996. On-Line Policy Improvement Using Monte-Carlo Search. In *NIPS*.

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

Toyer, S.; Trevizan, F.; Thiebaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *AAAI*.

Trevizan, F. W.; Thiébaux, S.; and Haslum, P. 2017. Occupation Measure Heuristics for Probabilistic Planning. In *ICAPS*.