Action Policy Testing with Heuristic-Based Bias Functions

Xandra Schuler[1], **Jan Eisenhut**[1], **Daniel Höller**[1], **Daniel Fišer**[1], **Jörg Hoffmann**[1,2]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
s8xaschu@stud.uni-saarland.de, {eisenhut,hoeller,danfis,hoffmann}@cs.uni-saarland.de

Abstract

When using action policies for sequential decision making, *testing* is a valuable tool to gain trust in their decisions. In particular, the action policies obtained via training deep neural networks are notoriously hard to verify and therefore need to be tested as black-box policies. Early approaches to *policy testing* aimed at finding environment conditions with problematic policy decisions. However, in these states, such decisions might be unavoidable, even for an optimal policy. Recently, a novel approach was introduced that instead looks for states where there provably exists a better decision. Such states are called (policy) *bugs*. While proving that a state is a bug is in general as hard as finding an optimal policy, several approaches have been proposed to successfully solve it in practice. They contain two sub-tasks that can be tackled separately: finding states prone to be bugs and proving that they actually are bugs. Here we deal with the former task and introduce two novel approaches for generating test states, which we call *loopiness bias* and *surface bias*. Both use techniques from heuristic search in classical planning to find states likely to cause the policy to behave suboptimally.

Introduction

Action policies based on neural networks have been highly successful. First in game playing (Mnih et al. 2013; Silver et al. 2016, 2018), but more recently also 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).

One strength of systems using such policies is that they can determine the next action to execute using a single evaluation of their policy on the current state of the environment leading to real-time decisions. However, such an approach raises concerns about (potentially fatal) undesirable behavior of the policy. Action policy testing is a natural way to gain trust in a policy. Early approaches on testing in sequential decision making control the behavior of the environment, steering the system into problematic areas (e.g., Dreossi et al. 2015; Akazaki et al. 2018; Koren et al. 2018; Ernst et al. 2019; Lee et al. 2020). However, when there is no way to avoid fatal decisions of the policy (even for an optimal policy), the policy is not to blame.

Recently, a novel framework for policy testing has been introduced (Steinmetz et al. 2022). The basic idea is to find environment states where the policy behaves suboptimally (with regard to some testing objective), i.e., where there exists a better decision to make. Such states are called *bugs* of the policy. While solving the problem of "bug detection" exactly obviously implies solving the problem of finding an optimal policy, several methods have been introduced that can identify policy bugs quite well in practice (Steinmetz et al. 2022; Eisenhut et al. 2023). The basic approach can be divided into two parts. In the first step, called *fuzzing*, a pool of states prone for being bugs is generated. In the second step, called *bug confirmation*, it needs to be proven that the tested policy behaves suboptimally on these states—the methods used for deciding whether the policy's decision on the given state is suboptimal are called *oracles*.

Both Steinmetz et al. (2022) and Eisenhut et al. (2023) focus on deterministic neural network policies built with the Action Schema Networks (ASNets) technique (Toyer et al. 2018, 2020) and use a very basic fuzzer for generating test states. For bug confirmation, Steinmetz et al. (2022) evaluated various oracles including a plan-improvement tool Aras (Nakhost and Müller 2010), a depth-first lookahead search repeatedly executing the input policy on different states in order to prove that better solution exists, and a couple of oracles specialized to invertible planning tasks. Eisenhut et al. (2023) follow on the work in the area of discrete-time Markov decision processes of Enişer et al. (2022) introducing *metamorphic* oracles. These oracles compare the policy behavior on pairs of states of which one is known to be easier to solve. So, when the policy performs worse on an easier state than on a harder state, a policy bug has been found. Eisenhut et al. (2023) uses an analogous idea in the area of classical (deterministic) planning leveraging the previous work on quantitative dominance functions (Torralba 2017). They show that such functions allow us to compare states in a domain-independent manner leading to metamorphic test oracles that can be successfully combined with the (search-based) oracles used by Steinmetz et al. (2022).

In this paper, we focus exclusively on the fuzzing step, i.e., we look for methods to find a pool of states likely to exhibit bugs. We build on the fuzzing technique introduced by Steinmetz et al.: a random walk starting at the initial state that can be steered towards more promising test states by so-called *bias functions*. A bias function takes a state and returns a number where higher numbers indicate more promising test states. We propose several new bias functions exploiting well-known heuristic functions from classical planning. We start with very simple bias functions based on the idea that the further away from the goal the given state is, the more likely it is the policy will fail. We end with more involved bias functions that compare the quality of a path induced by the policy with heuristic estimates to predict where the policy is behaving suboptimally. Our preliminary experimental evaluation shows encouraging results.

Background

An **FDR planning task** is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, I, G \rangle$. $\mathcal{V}$ is a finite set of **variables**, each variable $V \in \mathcal{V}$ has a finite **domain** $\mathrm{dom}(V)$. A **fact** $\langle V, v \rangle$ is a pair of a variable $V \in \mathcal{V}$ and one of its values $v \in \mathrm{dom}(V)$. A **partial state** p is a variable assignment over some variables $\mathrm{vars}(p) \subseteq \mathcal{V}$. We write $p[V]$ for the value assigned to the variable $V \in \mathrm{vars}(p)$ in the partial state p. Given a set of variables $U \subseteq \mathcal{V}$, $p[U]$ denotes a partial state p restricted to U. We also identify p with the set of facts contained in p, i.e., $p = \{\langle V, p[V] \rangle \mid V \in \mathrm{vars}(p)\}$. A partial state s is a **state** if $\mathrm{vars}(s) = \mathcal{V}$. The set of all states is denoted by $\mathcal{S}$ and the set of all partial states is denoted by $\mathcal{P}$. I is an **initial state**. G is a partial state called **goal**, and a state s is called a **goal state** if $G \subseteq s$.

$\mathcal{A}$ is a finite set of **actions**, each action $a \in \mathcal{A}$ is a triple $\langle \mathrm{pre}(a), \mathrm{eff}(a), \mathrm{cost}(a) \rangle$ of a **precondition**, **effect**, and **cost** respectively. Preconditions and effects are partial states and costs are non-negative real numbers. An action a is **applicable** in a state s if $\mathrm{pre}(a) \subseteq s$. Given a state s, $\mathcal{A}[s]$ denotes the set of all actions applicable in s. The **resulting state** of applying an applicable action a in a state s is the state $a[\![s]\!]$ where $a[\![s]\!][V] = \mathrm{eff}(a)[V]$ for every $V \in \mathrm{vars}(\mathrm{eff}(a))$, and $a[\![s]\!][V] = s[V]$ for every $V \in \mathcal{V} \setminus \mathrm{vars}(\mathrm{eff}(a))$.

A sequence of actions $\phi = \langle a_1, \dots, a_n \rangle$ is **applicable** in a state s_0 if there are states $s_1, \dots, s_n$ such that a_i is applicable in s_{i-1} and $s_i = a_i[\![s_{i-1}]\!]$ for $1 \leq i \leq n$. The resulting state of this application is $\phi[\![s_0]\!] = s_n$. Given an action sequence $\phi = \langle a_1, \dots, a_n \rangle$ applicable in a state s, the sequence $\langle s_0, s_1, \dots, s_n \rangle$ of the aforementioned states is called **intermediate state sequence** of ϕ. The cost of the action sequence $\phi = \langle a_1, \dots, a_n \rangle$ is defined as $\mathrm{cost}(\phi) = \sum_{i=1}^{n} \mathrm{cost}(a_i)$.

Given states s and t, a sequence of operators ϕ is called an ***s*-*t*-path** if ϕ is applicable in s and $\phi[\![s]\!] = t$. An s-t-path ϕ is called ***s*-plan** if t is a goal state. An I-plan is simply called a plan. An s-t-path (s-plan, plan) is called optimal if its cost is minimal among all s-t-paths (s-plans, plans).

We will need estimators for costs of optimal s-t-paths and s-plans. So, we define a **heuristic function** h as a function mapping a pair of a state and a partial state to a number or infinity (indicating unreachability of any goal state from the given state), i.e., $h \colon \mathcal{S} \times \mathcal{P} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$. Given two states s and t, the heuristic h estimates the cost of the optimal s-t-path. Given a state s, $h(s)$ denotes a shorthand for $h(s, G)$, i.e., $h(s)$ denotes a heuristic estimate for the cost of the optimal s-plan. $h^\star(s, t)$ denotes the cost of the optimal s-t-path, and $h^\star(s)$ denotes the cost of the optimal s-plan. All heuristics we consider here are safe, i.e., they return ∞ only if there is no s-t-path (or s-plan).

Action Policy

We consider deterministic state-dependent policies. A **policy** $\pi \colon \mathcal{S} \mapsto \mathcal{A} \cup \{\emptyset\}$ is a function mapping states to applicable actions or null ($\emptyset$) if there is no applicable action, i.e., for every state $s \in \mathcal{S}$ and policy π it holds that $\pi(s) = \emptyset$ if $\mathcal{A}[s] = \emptyset$ and $\pi(s) \in \mathcal{A}[s]$ otherwise. Given a policy π and a state s, the **run** of π on s, denoted by $\sigma^\pi(s)$, is a sequence of actions $\sigma^\pi(s) = \langle a_1, \dots, a_n \rangle$ applicable in s (with its intermediate state sequence $\langle s_0, \dots, s_n \rangle$) such that

(i) for every $i \in \{1, \dots, n\}$ it holds that $a_i = \pi(s_{i-1})$,

(ii) for every $i \in \{0, \dots, n-1\}$ it holds that s_i is not a goal state, and

(iii) for every $i, j \in \{0, \dots, n-1\}$ with $i \neq j$ it holds that $s_i \neq s_j$.

In other words, the run of π in s is the action sequence resulting from iteratively applying the policy π starting in the state s in such a way that the run terminates either when there is no applicable action, or the first goal state is reached, or the policy loops, i.e., it reaches some state for the second time. Clearly, $\sigma^\pi(s)$ is finite and unique.

The **cost** of the run $\sigma^\pi(s)$, denoted by $\mathrm{cost}^\pi(s)$, is defined as $\mathrm{cost}^\pi(s) = \mathrm{cost}(\sigma^\pi(s))$ if $\sigma^\pi(s)[\![s]\!]$ is a goal state, and $\mathrm{cost}^\pi(s) = \infty$ otherwise.

Given a policy π, state s_0, the run $\sigma^\pi(s_0) = \langle a_1, \dots, a_n \rangle$, its intermediate state sequence $\langle s_0, \dots, s_n \rangle$, and two numbers i, j such that $0 \leq i < j \leq n$, $\mathrm{cost}_{s_0}^\pi(s_i, s_j)$ denotes the cost of the subsequence of $\sigma^\pi(s_0)$ between s_i and s_j, i.e., $\mathrm{cost}_{s_0}^\pi(s_i, s_j) = \sum_{k=i+1}^{j} \mathrm{cost}(a_k)$. Note that computing $\mathrm{cost}_{s_0}^\pi(s_i, s_j)$ does not require knowing the whole policy run unless $s_j = s_n$. This allows to put a limit on the number of steps used to compute a policy run.

Action Policy Testing

The goal of action policy testing is to identify states where the given policy behaves suboptimally. Here, we adopt the testing framework introduced by Steinmetz et al. (2022). Given an FDR task Π and a policy π, a state $t \in \mathcal{S}$ is called a **bug** in π if $\mathrm{cost}^\pi(t) > h^\star(s)$. Note that this definition covers both the case (called "quantitative bug" by Steinmetz et al.) where π finds a suboptimal plan, i.e., both $\mathrm{cost}^\pi(t)$ and $h^\star(s)$ are finite, and the case (called "qualitative bug" by Steinmetz et al.) where π does not find a plan even though there exists one, i.e., $\mathrm{cost}^\pi(t) = \infty$ but $h^\star(s)$ is finite.

Policy testing requires (a) generating a pool $\mathcal{P} \subseteq \mathcal{S}$ of test states, and (b) running test oracles on the pool states $t \in \mathcal{P}$ to find bugs. Here, we are concerned with (a) only. For (b), we evaluate the best-performing oracle from the recent work of Eisenhut et al. (2023) denoted in their work as "BMO-100 + Aras/EHC".

This so-called bound maintenance oracle (BMO) maintains upper bounds $u(t)$ on $h^\star(t)$ across the states t it encounters during testing and attempts to propagate these bounds through state comparisons based on quantitative dominance functions (Torralba 2017). The underlying idea of how these bounds can be propagated is simple. Assume that we know that a previously encountered state s is solvable with a cost of at most 5 (i.e., $h^\star(s) \leq u(s) = 5$) and that t dominates s by cost 2 (i.e., $h^\star(t) \leq h^\star(s) - 2$). We can then easily conclude that 3 must be an upper bound on $h^\star(t)$, update $u(t)$ accordingly, and flag t as a bug if $h^\star(t) \leq u(t) < \mathrm{cost}^\pi(t)$. While the most basic BMO uses the cost of policy runs as the single initial source of upper bounds (as $h^\star(s) \leq \mathrm{cost}^\pi(t)$), any other sources such as search methods or external tools can be readily integrated. For that purpose, the "BMO-100 + Aras/EHC" oracle conducts a limited lookahead search (up to 100 steps) and runs enforced hill-climbing (EHC) or the plan-improvement tool Aras (Nakhost and Müller 2010) whenever a state cannot be confirmed to be a bug otherwise.

For generating pools of test states (test pools), we follow the approach used in previous works by using fuzzers. A **fuzzer** in general is a function that takes a state s and returns another state t such that t is a modification of s hopefully exhibiting a bug. In other words, fuzzers try to modify the input state so that it has features causing the given policy to fail or behave suboptimally.

Here, we use the fuzzer based on random walks with different bias functions proposed by Steinmetz et al. (2022, Algorithm 1). A **bias function** $\mathcal{B}\colon \mathcal{S} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ is a function that tries to assign higher values to states it deems closer to a bug state. The random-walk-based fuzzer with the given bias function $\mathcal{B}$ then works as follows: It starts in the initial state and performs a random walk of the maximal length L.

In each step of the random walk, the bias function $\mathcal{B}$ is evaluated on every successor state s and then the next state is selected randomly with probability proportional to $\mathcal{B}(s)$. If a safe heuristic h is a component of $\mathcal{B}$, we use h as a dead end detector, i.e., we ignore s if $h(s) = \infty$ as dead ends cannot be bugs. Conversely, if $\mathcal{B}$ requires running π and we determine that π fails on s, we assign a bias value of ∞.

Since running ASNet policies is very expensive, we need to make sure that the number of policy evaluations in each step of the random walk (that is, each state expansion) is bounded. To realize this, we limit both the number of steps we execute π in the computation of $\mathcal{B}(s)$ for each successor state s as well as for the entire state expansion, i.e., we maintain a budget for the number of policy evaluations across bias computations $\mathcal{B}(s)$. This simple mechanism for keeping the runtime at bay is a slight addition to the fuzzer proposed by Steinmetz et al. (2022). Also note that the original fuzzer additionally allows to filter out states based on the novelty measure (Lipovetzky and Geffner 2012), which we do not use here to make the experimental results easier to interpret.

The previous works (Steinmetz et al. 2022; Eisenhut et al. 2023) used two bias functions. The first one was the "blind" bias function $\mathcal{B}^0$ returning zero for every state resulting in successor states being selected uniformly at random.

The second bias function $\mathcal{B}^\pi$ returns the cost of the policy run $\mathrm{cost}^\pi(s)$ for each state s, i.e., the further the state is from a goal state according to the policy π, the more likely it is selected during the random walk. As we limit the number of steps we run π in each bias computation, we slightly deviate from the original definition in that $\mathcal{B}^\pi$ returns the cost of the respective *partial* policy run on s. Put differently, we added the option to abort running π, in which case $\mathcal{B}^\pi$ simply returns the accumulated cost of the partial run so far.

Here, we focus on different bias functions utilizing evaluation of heuristic functions.

Bias Functions Using Heuristics

The bias functions for generating test pools considered so far were only (a) the constant bias $\mathcal{B}^0$ resulting a standard random-walk selecting successor states uniformly at random, and (b) the policy bias $\mathcal{B}^\pi$ that favors states that the policy considers to be far from a goal state. The reasoning behind $\mathcal{B}^\pi$ is that further the policy thinks the state is from a goal state, the more likely it is the policy finds a suboptimal plan (or fails to find a plan altogether). The next obvious option following a similar reasoning is to use heuristic functions instead of π. That is, given a state s and a heuristic function h, $h(s)$ is an estimate of how far the state s is from a goal state. Therefore, we can use the given heuristic function directly as a bias function $\mathcal{B}^h = h$ and prefer states that are estimated to be further from a goal state.

The main advantage of using $\mathcal{B}^h$ is that evaluating heuristic functions typically requires far less time than (partially) running π (especially if π is based on ASNets). The clear disadvantage of $\mathcal{B}^h$ is that it does not take the actual behavior of π into account. On one hand, $\mathcal{B}^\pi$ prefers states that are further from the goal even if the policy π is actually right because the policy happens to behave optimally on those states. On the other hand, $\mathcal{B}^h$ prefers states that are actually further from the goal (assuming we trust the heuristic estimates), but disregards how the policy behaves on such states. What we would like is the combination of both, i.e., we want a bias function preferring states that the policy considers to be far from a goal state but that are actually not.

The main contribution of this paper is the introduction of two bias functions called **loopiness bias** and **surface bias**. Both utilize heuristics to steer the pool generation towards states where the policy behaves suboptimally by comparing the distance between two states according to the path taken by the policy with heuristic estimates of the distance between these states.

For the rest of this section, let π denote a policy, let h denote a safe heuristic, let s denote a state, let $\langle a_1, \dots, a_n \rangle$ denote a *partial* run of π on s, i.e., the first n steps on the policy run $\sigma^\pi(s)$, and let $\langle s_0, \dots, s_n \rangle$ denote the respective intermediate state sequence. We assume $h(s) < \infty$ and $n \geq 1$ as otherwise the fuzzer would not invoke the bias computation for s altogether.

Loopiness Bias

The loopiness bias is defined as follows:

$$\mathcal{B}_h^{\mathrm{loop}}(s) = \max(0, \max_{0 \leq i < j \leq n} (\mathrm{cost}_s^\pi(s_i, s_j) - h(s_i, s_j)))$$

For every pair of states s_i and s_j such that $i < j$, we compute the difference between the cost of reaching s_j from s_i using π (i.e., the distance between s_i and s_j according to π) and the distance between the two states according to the given heuristic function. $\mathcal{B}_h^{\mathrm{loop}}(s)$ is then the maximum of these differences if this is positive or 0 otherwise.

Provided that π is executed completely, it is easy to see that for an optimal π and $h^\star$, $\mathcal{B}_{h^\star}^{\mathrm{loop}}(s)$ is 0 for every state s. However, assuming we trust our heuristic function h, $\mathrm{cost}^{\pi}_{s}(s_i, s_j)$ being larger than $h(s_i, s_j)$ is indicative of π behaving suboptimally as h is indicating there exists a cheaper s_i-s_j-path than the one taken by π. In extreme cases where $\mathrm{cost}^{\pi}_{s}(s_i, s_j)$ is large and $h(s_i, s_j)$ is very small, the behavior of the policy could be interpreted as almost going in loops as it takes a long detour from s_i to s_j even though these states are close to each other.

Surface Bias

The surface bias is defined similarly to the loopiness bias:

$$\mathcal{B}^{\mathrm{surf}}_{h}(s) = \max(0, \max_{0 \le i < j \le n} (\mathrm{cost}^{\pi}_{s}(s_i, s_j) - (h(s_i) - h(s_j))))$$

The main difference is the way the heuristic function is used in order to measure the distance between states. While the loopiness bias uses the direct measure $h(s_i, s_j)$, the surface bias measures the distance indirectly by $h(s_i) - h(s_j)$. That is, rather than focusing on the cost of the s_i-s_j-path, the surface bias compares the (estimated) costs of the s_i-plan and s_j-plan. Moreover, in contrast to $h(s_i, s_j)$, $h(s_i) - h(s_j)$ can also be negative, in which case the heuristic is signaling that the policy is not even progressing towards the goal.

This approach is somewhat reminiscent of the state space surface analysis (e.g., Hoffmann 2003, 2011). Assuming h is a good estimator, we expect $h(s)$ to decrease by relatively the same amount as the cost of the path π takes while progressing through the state space, unless, of course, π is behaving suboptimally. Or put differently, if the policy makes a lot of steps ($\mathrm{cost}^{\pi}_{s}(s_i, s_j)$ is large), but does not get much closer to a goal state ($h(s_i) - h(s_j)$ is small or even negative), then we interpret such behavior as likely suboptimal.

Note that these interpretations are obviously subject to inaccuracies in the heuristic function h itself. In the ideal setting, $h^{\star}$ should be used in both the loopiness and the surface bias, to exclude this source of inaccuracy. Yet, that will typically not be possible in practice so that we need to use a practical proxy for $h^{\star}$ via standard heuristic functions h. To the extent possible, we evaluate in our experiments to which degree the replacement of $h^{\star}$ with h affects the quality of our fuzzing biases.

Preliminary Experimental Evaluation

All proposed methods were implemented in C++ and evaluated on a cluster with Intel E5-2660 processors. We test the ASNets policies (Toyer et al. 2018, 2020) on the same dataset as Eisenhut et al. (2023).

The experiments were conducted in two phases. In the first phase, the pool $\mathcal{P}$ containing up to 100 states was generated using different bias functions with one hour time limit and 16 GB memory limit. We ran π only for at most 50 steps in each invocation of a bias function and limited the overall number of steps to 200 per state expansion in the fuzzer.

In the second phase, the "BMO-100 + Aras/EHC" oracle (Eisenhut et al. 2023) was run on the pool states. The dominance functions of the oracle were precomputed with a time limit of four hours. The oracle itself was run with limits of two hours and 16 GB. Compared to the experimental setting used by Eisenhut et al., we used smaller time limits.

Domain	Σ	Pool size: Number of pool states (avg over domain)							
		$\mathcal{B}^{0}$	$\mathcal{B}^{\pi}$	$\mathcal{B}^{h^{\mathrm{ff}}}$	$\mathcal{B}^{\mathrm{surf}}_{h^{\mathrm{ff}}}$	$\mathcal{B}^{\mathrm{loop}}_{h^{\mathrm{ff}}}$	$\mathcal{B}^{h^{\star}}$	$\mathcal{B}^{\mathrm{surf}}_{h^{\star}}$	$\mathcal{B}^{\mathrm{loop}}_{h^{\star}}$
Blocks	24	**100.0**	42.0	**100.0**	40.4	50.1	1.0	1.0	7.7
Gripper	35	**95.0**	94.8	94.8	92.8	**95.0**	29.3	28.4	30.0
MBlocks	6	**100.0**	65.8	**100.0**	42.0	69.0	34.7	33.7	27.7
Satellite	16	**100.0**	44.7	**100.0**	36.8	40.6	30.1	19.4	23.4
Storage	7	**92.4**	73.1	90.4	73.4	74.9	78.6	53.3	74.6
Transport	24	**100.0**	96.4	**100.0**	73.4	95.2	66.7	63.7	66.2
VisitAll	19	**95.7**	29.2	95.5	30.4	29.1	1.7	1.9	4.3

Table 1: Number of generated pool states (average over domain, 100 is maximum). Σ denotes the number of tasks.

We compare the constant bias $\mathcal{B}^{0}$ and the policy bias $\mathcal{B}^{\pi}$ with our heuristic-based biases. We use the FF heuristic h^{ff} (Hoffmann and Nebel 2001) and $h^{\star}$ implemented as A* search with the LM-Cut heuristic h^{lmc} (Helmert and Domshlak 2009). Namely, we evaluate the bias functions $\mathcal{B}^{h^{\mathrm{ff}}}$ and $\mathcal{B}^{h^{\star}}$ that simply return the heuristic value for the given state, and the more elaborate surface and loopiness biases $\mathcal{B}^{\mathrm{surf}}_{h^{\mathrm{ff}}}$, $\mathcal{B}^{\mathrm{surf}}_{h^{\star}}$, $\mathcal{B}^{\mathrm{loop}}_{h^{\mathrm{ff}}}$, and $\mathcal{B}^{\mathrm{loop}}_{h^{\star}}$. For each heuristic h, especially for $h^{\star}$, we expected $\mathcal{B}^{\mathrm{loop}}_{h}$ to be slower than $\mathcal{B}^{\mathrm{surf}}_{h}$ because $\mathcal{B}^{\mathrm{loop}}_{h}$ requires evaluating h between all pairs of states whereas $\mathcal{B}^{\mathrm{surf}}_{h}$ only needs to compute h for each state once. For this reason, we restricted the loopiness bias to only k randomly sampled intermediate states of the (partial) policy run, and we set k to the square root of the length of the (partial) policy run. For h^{ff}, this restriction is probably not strictly necessary (as long as policy evaluations are much more expensive). An advantage of applying it here regardless is that it allows for a cleaner comparison between $\mathcal{B}^{\mathrm{loop}}_{h^{\mathrm{ff}}}$ and $\mathcal{B}^{\mathrm{loop}}_{h^{\star}}$.

We tried using h^{lmc} as an alternative to h^{ff}. However, these experiments indicated that there is not much difference between using h^{ff} and h^{lmc}, which is not surprising as both methods are based on extracting relaxed plans. We therefore decided to consider only h^{ff}, focusing on different kinds of biases rather than on applying different heuristic functions.

Pool Sizes

Table 1 shows the average number of test states over all tasks within each domain. $\mathcal{B}^{0}$ and $\mathcal{B}^{h^{\mathrm{ff}}}$ generate the highest number of states within the time limit because they are by far the fastest of the tested bias functions. We would expect that any bias including $h^{\star}$ generates considerably less test states than any bias not including $h^{\star}$, as computing $h^{\star}$ involves running an optimal planner. This is indeed mostly the case. For instance, for all biases including a heuristic, the variant using h^{ff} generates more states than the variant using $h^{\star}$. The only exception is the Storage domain where $\mathcal{B}^{\pi}$ generates slightly less test states on average then $\mathcal{B}^{h^{\star}}$ indicating that evaluating ASNets policies is sometimes slower than solving the task optimally.

$\mathcal{B}^{\pi}$, $\mathcal{B}^{\mathrm{surf}}_{h^{\mathrm{ff}}}$, and $\mathcal{B}^{\mathrm{loop}}_{h^{\mathrm{ff}}}$ generate roughly similar numbers of test states in most domains. However, there are significant differences in some domains, e.g., in Transport $\mathcal{B}^{\mathrm{surf}}_{h^{\mathrm{ff}}}$ generates more than 20 states less on average than $\mathcal{B}^{\pi}$ and

Domain	T	Oracle coverage				
		$\mathcal{B}^0$	$\mathcal{B}^\pi$	$\mathcal{B}^{h^{\text{ff}}}$	$\mathcal{B}^{\text{surf}}_{h^{\text{ff}}}$	$\mathcal{B}^{\text{loop}}_{h^{\text{ff}}}$
Blocks	402	19.7	19.4	19.4	21.6	**31.6**
Gripper	2442	95.2	95.1	94.9	95.3	**95.5**
MBlocks	252	**8.3**	2.0	4.4	0.8	1.6
Satellite	535	75.0	92.3	61.9	**94.2**	93.8
Storage	491	35.6	64.2	41.1	**67.4**	56.0
Transport	1432	70.4	82.3	71.9	87.4	**92.2**
VisitAll	449	84.0	90.4	83.7	93.1	**93.3**

Table 2: Oracle coverage for biases not using $h^\star$. T is the overall number states considered when computing oracle coverage (sum over domain, same for all biases). Oracle coverage is the percentage of states in the respective pool set (of size T) identified as bugs.

$\mathcal{B}^{\text{loop}}_{h^{\text{ff}}}$. While running the policy typically accounts for most of the runtime when computing either of these three biases, the runtime of a bias computation obviously also depends on the state for which we compute the bias. As we use biases to steer the fuzzer into certain regions of the state space, the runtime of invocations of a bias function might thus very well depend on the result of previous bias invocations.

Oracle Coverage

In order to assess the quality of our biases, we run the oracle on the respectively generated pool states and count the number of pool states the oracle classifies as bugs. We refer to the percentage of tested pool states (across the entire domain) classified as bugs as *oracle coverage*.

When comparing a set of biases with respect to oracle coverage, an inherent problem is that this measure can be distorted if different biases generate different numbers of pool states for different instances. For example, it might happen that a bias $\mathcal{B}_1$ generally generates much better test states than $\mathcal{B}_2$ but generates more test states in instances where the policy performs better (i.e., in instances where there are simply less bugs to be found). As a result, the worse bias $\mathcal{B}_2$ that simply generates less states in these instances could still end up with a higher oracle coverage.

For this reason, we measure the oracle coverage with the same number of pool states across all biases for each problem instance, e.g., if we have 100 pool states for bias $\mathcal{B}_1$ in a problem instance X but only 50 for bias $\mathcal{B}_2$, we consider only the first 50 pool states of $\mathcal{B}_1$ when computing the oracle coverage for X.

Tables 2 and 3 both compare a subset of the biases based on oracle coverage. The setup is the same, only the selected biases are different. The T column shows the overall number of considered states for computing the coverage. As we take the minimal number of test states (for each instance separately) over all biases participating in the respective comparison, the T values are not the same in both tables.

Table 2 shows oracle coverage for the biases intended for practical use, i.e., the ones not requiring to evaluate $h^\star$. The oracle coverage of $\mathcal{B}^{h^{\text{ff}}}$ is not significantly better or sometimes even worse than $\mathcal{B}^0$. So, it seems using heuristic estimates alone does not lead to more bug states than a simple random walk choosing successor states uniformly at random. In contrast, either the surface bias $\mathcal{B}^{\text{surf}}_{h^{\text{ff}}}$ or the loopiness bias $\mathcal{B}^{\text{loop}}_{h^{\text{ff}}}$ reach the highest oracle coverage in all domains except for MBlocks. More precisely, $\mathcal{B}^{\text{surf}}_{h^{\text{ff}}}$ is the best performing bias in Satellite and Storage, whereas $\mathcal{B}^{\text{loop}}_{h^{\text{ff}}}$ is the best performing bias in Blocks, Gripper, Transport, and VisitAll. However, while the difference between $\mathcal{B}^{\text{surf}}_{h^{\text{ff}}}$, $\mathcal{B}^{\text{loop}}_{h^{\text{ff}}}$, and $\mathcal{B}^\pi$ is significant in some domains (such as Transport), there are also domains where the oracle coverage is virtually the same (such as Gripper). Yet, it seems that the surface and loopiness biases are an informative mixture of information about how the policy behaves and information about its traversed states from h^{ff} as another source. Also note that in three of our domains the oracle coverage for $\mathcal{B}^\pi$ is already higher than 90% so that reaching a substantial improvement with a more informed bias is hardly possible.

Domain	T	Oracle coverage					
		$\mathcal{B}^{h^{\text{ff}}}$	$\mathcal{B}^{\text{surf}}_{h^{\text{ff}}}$	$\mathcal{B}^{\text{loop}}_{h^{\text{ff}}}$	$\mathcal{B}^{h^\star}$	$\mathcal{B}^{\text{surf}}_{h^\star}$	$\mathcal{B}^{\text{loop}}_{h^\star}$
Gripper	910	90.2	90.4	91.5	88.8	**94.7**	91.9
MBlocks	118	4.2	1.7	1.7	29.7	**78.8**	1.7
Satellite	261	55.9	92.7	92.3	61.3	**93.9**	89.3
Storage	345	34.5	56.8	40.6	31.9	**62.0**	40.0
Transport	756	54.2	77.0	86.0	51.6	**93.9**	82.0

Table 3: Oracle coverage for biases using $h^\star$ or h^{ff}. Same setup as in Table 2. The Blocks and VisitAll domains are excluded as the biases based on $h^\star$ do not generate enough states.

We now come back to the question of to which degree the replacement of $h^\star$ with h^{ff} affects the quality of our fuzzing biases. Table 3 shows oracle coverage for $\mathcal{B}^{h^\star}$, $\mathcal{B}^{\text{surf}}_{h^\star}$, and $\mathcal{B}^{\text{loop}}_{h^\star}$ as well as for their practical proxies $\mathcal{B}^{h^{\text{ff}}}$, $\mathcal{B}^{\text{surf}}_{h^{\text{ff}}}$, and $\mathcal{B}^{\text{loop}}_{h^{\text{ff}}}$. $\mathcal{B}^{\text{surf}}_{h^\star}$ reliably dominates $\mathcal{B}^{\text{surf}}_{h^{\text{ff}}}$ in terms of oracle coverage across all domains, but $\mathcal{B}^{h^\star}$ and $\mathcal{B}^{h^{\text{ff}}}$ as well as $\mathcal{B}^{\text{loop}}_{h^\star}$ and $\mathcal{B}^{\text{loop}}_{h^{\text{ff}}}$ seem to be on-par with each other in most domains. The most notable differences are in MBlocks where $\mathcal{B}^{h^\star}$ has much higher oracle coverage than $\mathcal{B}^{h^{\text{ff}}}$, and in Transport where, unexpectedly, $\mathcal{B}^{\text{loop}}_{h^{\text{ff}}}$ beats $\mathcal{B}^{\text{loop}}_{h^\star}$ by four percentage points. Overall h^{ff} seems to be a better proxy for $h^\star$ in the case of heuristic and loopiness biases, than in the case surface bias.

Conclusion

With the proliferation of learned action policies in sequential decision making, the question how to meaningfully test such policies is becoming very relevant. Our work makes a technical contribution to this, in the specific context of exploring fuzzing biases for test-state generation within previously proposed methods in planning (Steinmetz et al. 2022; Eisenhut et al. 2023). We introduce two new biases, the loopiness and the surface bias, which combine information about the path taken by the policy with heuristic estimates. Judging by our current experiments, these biases are informative in many domains, and can soundly beat previous fuzzing biases in some domains.

For future work, it remains to consider additional other kinds of learned policies, beyond ASNets, to broaden the empirical basis for our study which generalizes, after all, across arbitrary policies. The approach by Stahlberg et al. (2022; 2022) is a natural candidate for this step.

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.

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.

Eisenhut, J.; Torralba, Á.; Christakis, M.; and Hoffmann, J. 2023. Automatic Metamorphic Test Oracles for Action-Policy Testing. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS'23)*, Accepted.

Enişer, 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 *Proceedings of the 16th International Conference on Quantitative Evaluation of Systems (QEST'19)*, 165–181.

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.

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.; 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.

Hoffmann, J. 2003. *Utilizing Problem Structure in Planning: A Local Search Approach*, volume 2854 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Hoffmann, J. 2011. Analyzing Search Topology Without Running Any Search: On the Connection Between Causal Graphs and h^+. *Journal of Artificial Intelligence Research*, 41: 155–229.

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.

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In Raedt, L. D., ed., *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI'12)*, 540–545. Montpellier, France: IOS Press.

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

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. AAAI press.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529: 484–503.

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.

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 Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24 2022*, 629–637.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning Generalized Policies without Supervision Using GNNs. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning (KR'22)*.

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 *Proc. IJCAI'17*, 4426–4432.

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 McIlraith, S.; and Weinberger, K., eds., *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*. AAAI Press.