Using Action-Policy Testing in RL to Reduce the Number of Bugs

Hasan Ferit Eniser[1], **Songtuan Lin**[2], **Nicola Müller**[3], **Anastasia Isychev**[4], **Valentin Wüstholz**[5], **Isabel Valera**[2], **Jörg Hoffmann**[2,3], **Maria Christakis**[4]
[1] MPI-SWS, Germany
[2] Saarland Informatics Campus, Saarland University, Germany
[3] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
[4] TU Wien, Austria
[5] Consensys, Austria
hfeniser@mpi-sws.org, {songtuan.lin, ivalera, hoffmann}@cs.uni-saarland.de, nicola.mueller@dfki.de, {anastasia.isychev, maria.christakis}@tuwien.ac.at, valentin.wustholz@consensys.net

Abstract

Reinforcement learning is becoming ever more prominent in solving combinatorial search problems, in particular ones where states are images. Prior work has devised action-policy testing methodology, that identifies so-called bug states where policy performance is sub-optimal. Here we show how to leverage this methodology during the RL process, using action-policy testing to find bugs and injecting those as alternate start states for the training runs. Running experiments across six 2D games, we find that our testing-guided training often achieves similar expected reward while reducing the number of bugs.

Introduction

In the context of decision-making, an *action policy* refers to a mapping between *states* and *actions*. Given a state (which is an abstraction of an environment), a policy outputs an action that is to be taken in that state. Recently, policies represented by neural networks (NNs) have drawn increasing attention in problems involving decision-making, e.g., in games (Mnih et al. 2015; Silver et al. 2016, 2018) and in automated planning tasks (Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020; Karia and Srivastava 2021; Ståhlberg, Bonet, and Geffner 2022a,b; Wang and Thiébaux 2024).

One important aspect of a policy we are usually concerned with is whether it behaves wrongly in some states. For instance, in the scenario of autonomous driving, an incorrect behavior of a policy in a state may lead to a crash. In general, we call such a state a bug state.

Existing literature uses *testing* to identify bug states, see, e.g., the work by Dreossi et al. (2015), Akazaki et al. (2018), and Koren et al. (2018). For an overview, we refer to the work by Corso et al. (2021). This line of work focuses on the *safety* of a policy, that is, it is concerned with bug states where policies' incorrect behaviors cause unsafe situations.

We consider bug states in a more general sense. In this paper, we regard a state where a policy performs *sub-optimally* as a bug state. We leverage the π-fuzz testing framework (Eniser et al. 2022) to identify such bug states and propose an approach fusing Deep Reinforcement Learning training with testing to improve a policy's performance on bug states.

Our contribution is twofold. First, we introduce a variant of the *test oracle* in π-fuzz, which identifies bug states where a policy performs sub-optimally. Second, we propose an approach incorporating testing into Reinforcement Learning training which injects bug states found by testing into training runs as start states.

We evaluate our approach on 6 domains (environments), ranging from 2D games to simplified autonomous driving scenarios. The experimental results show that our approach can reduce the number of bug states while still maintaining similar policy performance compared with standard training.

Background

We consider *Markov Decision Processes* (MDPs) as the underlying decision-making framework, on which Reinforcement Learning approaches are deployed.

A MDP $\mathcal{M}$ is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{P}_I, \gamma)$. $\mathcal{S}$ is a set of *states*. $\mathcal{A}$ is a set of *actions*. Both $\mathcal{S}$ and $\mathcal{A}$ can be *infinite*. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{D}(\mathcal{S})$ is a function where $\mathcal{D}(\mathcal{S})$ is the set of all probability distributions over $\mathcal{S}$. Intuitively, $\mathcal{T}$ captures how applying an action in a state can change that state to another. For instance, suppose that $\mathcal{T}(s, a) = \mathcal{P}$ where $\mathcal{P} \in \mathcal{D}(\mathcal{S})$ is a probability distribution over $\mathcal{S}$, then $\mathcal{P}(s')$ with $s' \in \mathcal{S}$ defines the probability of obtaining the state s' by applying the action a in s. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a function defining the *reward* that can be obtained by applying an action in a state. $\mathcal{P}_I$ is a probability distribution over $\mathcal{S}$ which defines the probability of a state being the *initial state*. γ with $0 < \gamma < 1$ is called a decay factor.

A *policy* for a MDP is a function $\pi : \mathcal{S} \to \mathcal{A}$ deciding which action should be taken for each state. A policy π induces a *state-action trace* $\tau = \langle s_0\, a_0\, s_1\, a_1\, s_2\, a_2 \cdots \rangle$ where s_0 is the initial state with $\mathcal{P}_I(s_0) > 0$, and for each $t \geq 0$, $a_t = \pi(s_t)$, and $\mathcal{P}_{t+1}(s_{t+1}) > 0$ with $\mathcal{T}(s_t, a_t) = \mathcal{P}_{t+1}$ (i.e., the action a_t changes the state s_t to s_{t+1}). A state-action trace is also called an *episode*. The total reward $r(\tau)$ associated with an episode τ is $r(\tau) = \sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t)$ where γ is the decay factor. The presence of γ ensures that $r(\tau)$ will converge to a real number despite that τ is infinite.

Since applying an action in a state could result in different next states, a policy π may induce different state-action traces. Therefore, we are more interested in the average reward over all possible state-action traces governed by a policy π, i.e., $\mathbb{E}_{\tau\sim\pi}[\![r(\tau)]\!]$. For convenience, we call this the *expected reward* of π. For an MDP, we want to find a policy whose expected reward is as high as possible.

One way to find such a policy is by Reinforcement Learning (RL), especially Deep Reinforcement Learning (DRL), which has drawn increasing attention in the last decade. We briefly present the general workflow of DRL here and omit the details, because we could basically view the training and optimization process of DRL as a black box in this paper.

DRL starts with randomly initializing a policy π. The policy π is then updated iteratively. In each iteration, a state-action trace τ is first generated (i.e., sampled) according to the current π. After that, π is optimized (i.e., updated) based on the total reward of τ (and potentially on the total rewards of the state-action traces generated in previous iterations, depending on different variants of DRL approaches). The training stops when the number of iterations reaches the limit.

Next we present the π-fuzz testing framework proposed by Eniser et al. (2022), which is used to identify *bug states* for policies. Formally speaking, we can define a bug state as follows: Let π be a policy, $\mathcal{S}$ a set of states, and P a desirable property pertaining to π and states in $\mathcal{S}$ (e.g. P could state that π does not crash into an obstacle). A state $s \in \mathcal{S}$ is a bug state for π if P can be made true in s but is not true under π. For identifying bug states, the hard part is how to decide whether P can be made true in s, because that in general necessitates optimal planning.

The π-fuzz testing framework addresses this issue by exploiting the notion of *state relaxations*. For two states s' and s, we say that s' *relaxes* s if s' is easier to solve than s, in the sense that any solution for s also works for s'. State relaxations provide a means for identifying bug states because, for the properties P we are interested in, if P is true in s then it must also be true in s'; and therefore, if under π P is true in s but is not true in s', then s' is a bug state. Eniser et al. (2022) exploited state relaxations to identify bug states related to *safety*. In their context, a state s' is a bug in π if running π on s' enters a state that satisfies a *failure condition* ϕ, even though this could be avoided. Their oracle checks whether for a pair of state s and relaxed state s', π enters a failed state on s' but not on s, in which case s' is a bug.

The π-fuzz framework consists of two components: a *fuzzer* and a *test oracle*. The fuzzer generates a *pool* of pairs (s', s) of states where s' relaxes s. The test oracle then runs π on all state pairs in the pool, checking whether s' can be proved to be a bug as described above.

Testing Methods

In this paper, we adapt the π-fuzz framework to deal with bug states in a slightly more general sense. Along the lines of policy testing work in classical planning (Steinmetz et al. 2022; Eisenhut et al. 2023, 2024) we say that a state s' is a bug state for a policy π if π performs *sub-optimally* in s'. In our case, this means that the expected reward of π on s' could be *higher*. Like Eniser et al. (2022), we exploit state relaxations to identify such bug states. Concretely, for any

Algorithm 1 The procedure of Testing-Guided training.

```
1: $\pi \leftarrow$ randomly initialized policy
2: $f \leftarrow$ test frequency
3: $\triangleright$ $\mathcal{C}$ is a multi-set
4: $p^* \leftarrow 0, \mathcal{C} \leftarrow \emptyset, D \leftarrow$ FUZZER()
5: for $i = 1, \ldots, L$ do                        $\triangleright$ training iterations
6:     $\triangleright$ testing with the frequency $f$
7:     if $i \,\%\, (L \times f) = 0$ then
8:         $\mathcal{C} \leftarrow \mathcal{C} \cup$ TESTORACLE$(\pi, D)$
9:     if BERNOULLI$(p^*) = 1$ then
10:        $\tau \leftarrow$ SAMPLETRACE$(\pi, s_0 \in \mathcal{C})$
11:    else $\tau \leftarrow$ SAMPLETRACE$(\pi, s_0 \sim \mathcal{P}_I)$
12:    $\triangleright$ standard RL optimization using back propagation
13:    $\pi \leftarrow$ UPDATEPOLICY$(\tau)$
14:    $\triangleright$ $p^*$ is updated on-the-fly
15:    $p^* \leftarrow$ UPDATEPROBABILITY$(\pi)$
16: return $\pi$
```

two states s' and s, if s' relaxes s then the expected reward of π on s' should be at least as good as that on s, i.e.,

$$\mathbb{E}_{\tau\sim\pi}[\![r(\tau)|s_0 = s']\!] \geq \mathbb{E}_{\tau\sim\pi}[\![r(\tau)|s_0 = s]\!] \qquad (1)$$

If s' relaxes s and Ieq. 1 does *not* hold, then s' is a bug state.

Fuzzer We could directly adopt the implementation of the fuzzer by Eniser et al. (2022) for the purpose of generating a pool of pairs (s', s) of states where s' relaxes s.

Test Oracle Given two states s' and s with s' being a relaxed state of s and a policy π, our test oracle shall be able to check whether Inequality 1 holds. One problem here is that it is practically infeasible to compute the expected reward of π on s' (and on s). To address this issue, we run the policy N times on both s and s' where N is a number decided by us. We use τ_i and τ'_i to denote the ith state-action trace obtained by running π on s and s', respectively. If $r(\tau'_i) < r(\tau_i)$ for all $1 \leq i \leq N$, then we regard s' as a bug state.

Testing-Guided RL

The presented testing method identifies bug states for a policy, which, in our context, are states where the policy performs sub-optimally. We further want to improve the performance of the policy on these bug states.

Recall that in a DRL training process, a policy is updated (i.e., its performance is improved) by sampling state-action traces whose start states follow the initial state distribution. Hence, the performance of a policy on bug states can be improved by sampling state-action traces whose start states are drawn from those bug states. A similar idea has been exploited before to improve the safety of a policy through targeted selection of start states (Gros et al. 2023, 2024).

More concretely, during DRL training, when a new state-action trace is going to be sampled, we could set a probability to control whether the initial state of this trace is drawn from the initial state distribution or from bug states. This setting thus ensures that the start states of the state-action traces sampled for training alternate between normal initial states and bug states, improving the policy's performance on both types of states simultaneously.

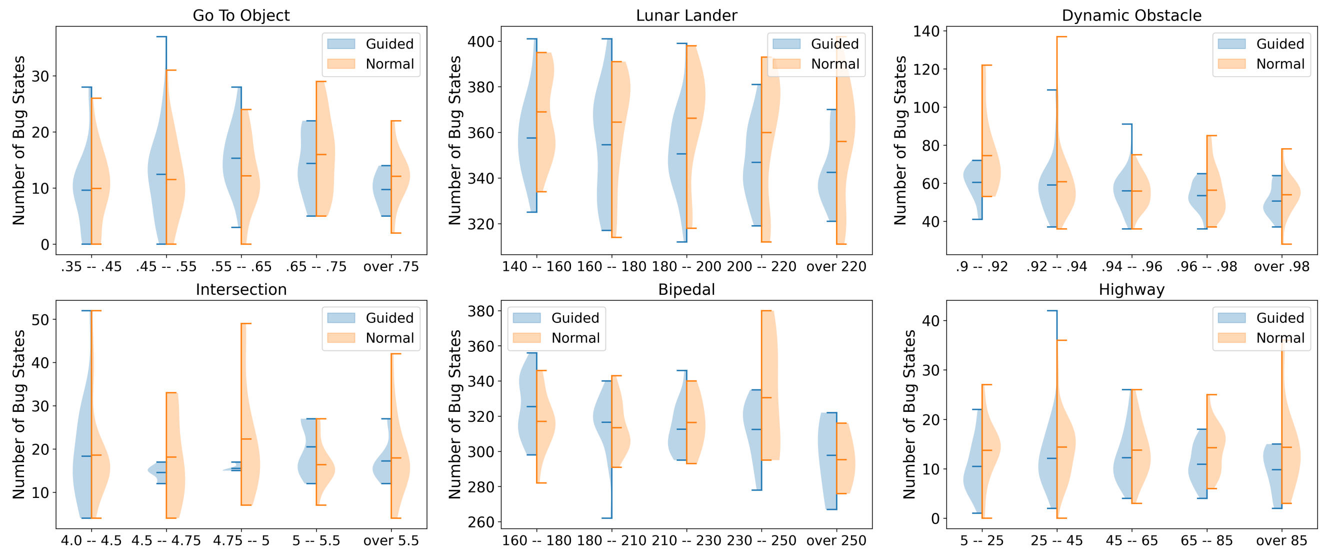

Figure 1: Distributions of number of bug states within reward intervals, for the tested policies in 5 runs of training as per Alg. 1.

The realization of this idea is shown in Alg. 1. We first use the fuzzer to generate a pool D of state pairs (s', s) *before* training begins where s' relaxes s from which bug states are identified. During training, we perform testing (using the test oracle) with a certain frequency. The bug states returned in every test are added to a collection $\mathcal{C}$, which is initialized to an empty set. When a new state-action trace is going to be sampled (for updating the policy), the start state of this trace is either drawn according to the initial state distribution or from the collection $\mathcal{C}$ of bug states (if $\mathcal{C}$ is not empty).

Performing testing with a certain frequency during training is to keep track of newly appeared bug states. Along with the process of updating the policy, a state in the pool which was not a bug state may become one. Furthermore, we want to emphasize that the collection $\mathcal{C}$ is a *multi-set*, that is, a bug state could appear multiple times in $\mathcal{C}$. In particular, if a state indeed appears multiple times in $\mathcal{C}$, i.e., it is identified as a bug state in multiple rounds of test, then it has higher chance of being drawn as the start state of a state-action trace.

The last important aspect of our approach is that, the probability that decides whether the start state of a trace is drawn from $\mathcal{C}$ or according to $\mathcal{P}_I$ is computed *on-the-fly*. The reason for this is that, intuitively speaking, if this probability is too high, the training process would focus too much on improving the performance of the policy on bug states, which could cause a decrease in the performance on ordinary initial states. Generally speaking, this probability is computed according to the total rewards of state-action traces sampled according to the current policy. We want the probability to be slightly high if the total rewards of those sampled traces do not change too much.

This change rate of rewards can be captured numerically. Let's say, *wlog*., we consider the rewards $r_1, \cdots, r_k$ of k traces. We first normalize every reward by letting $\hat{r}_i = r_i/M$ where $M = \max\{r_i \mid 1 \leq i \leq k\}$ is the maximal reward. The purpose of this step is just to make the latter step of computing the probability p^* easier. For each $1 \leq i \leq k$, we define $\Delta_i = |\hat{r}_{i+1} - \hat{r}_i|$, which captures how much the reward changes between two consecutively sampled sequences. We can compute the average change rate of those rewards as:

$$\boldsymbol{\Delta} = \frac{1}{k-1} \sum_{i=1}^{k-1} \Delta_i$$

The probability p^* can then be obtained accordingly: $p^* = \alpha(1 - \boldsymbol{\Delta})$ where $0 \leq \alpha \leq 1$ is a parameter decided by us. If the rewards of those traces change sharply, $\boldsymbol{\Delta}$ would then be large, reducing the probability p^*. Conversely, if that is not the case, then $\boldsymbol{\Delta}$ would be small, increasing p^*.

Experiments

We evaluated our approach on 6 domains, ranging from 2D games to simplified autonomous driving scenarios. The primary objective of our experiments is to evaluate whether our approach can reduce the number of bug states for a policy.

To this end, we ran a standard RL training approach (we ran PPO (Schulman et al. 2017) on all domains except for Go To Object on which we used DQN (Mnih et al. 2013)) and our testing-guided training approach on each domain. We set the test frequency to 0.02 in our approach. For both training approaches, we ran 5 trials. The reason for running multiple trials is to reduce the inherent noise during training. For each test, we record the number of bug states found and the expected reward of the corresponding policy. We set $\alpha = 0.5$ in all domains, except in Go To Object where $\alpha = 0.35$ as larger values could lead to a reduction in the rewards of policies on ordinary initial states.

Fig. 1 shows violin plots for the distribution of the number of bug states in the policies tested by Alg. 1 during training. We group the policies by their reward performance, which we bin into given intervals (the number of policies in each bin is shown in Fig. 2). We show intervals for reasonably

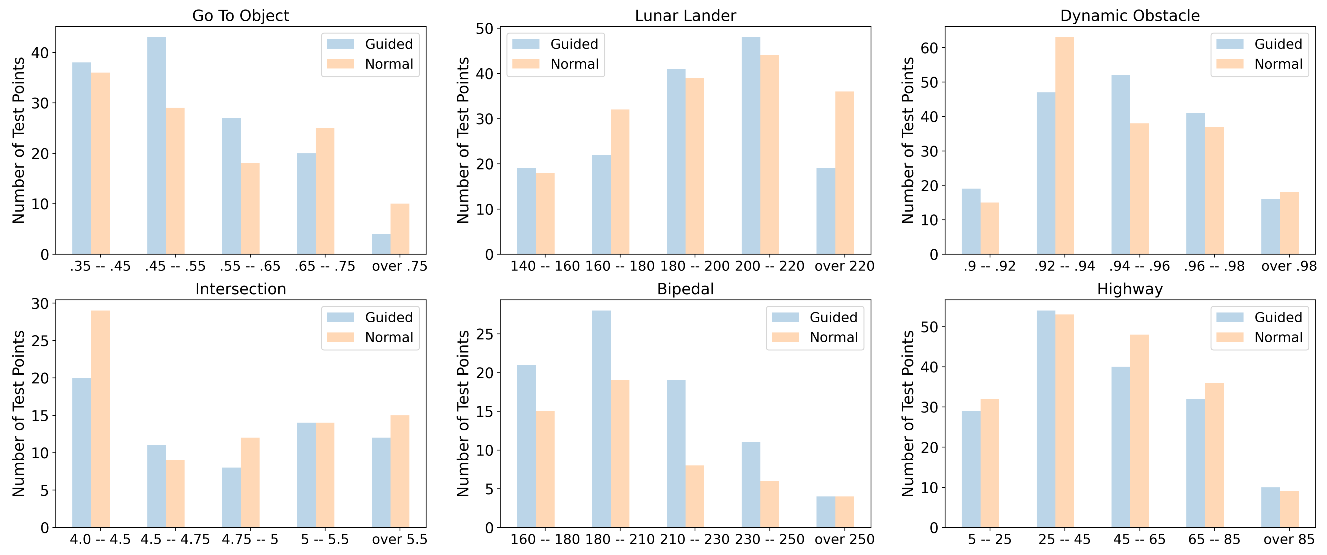

Figure 2: Number of policies within each reward-interval bin underlying Fig. 1.

well-performing polices only, as low-performance policies are not of interest. In what follows, for each domain we give an outline and summarize our results.

Go To Object is a Minigrid (Chevalier-Boisvert et al. 2023) domain. We generated a pool with 500 state pairs. The results are modestly positive, with a mixed picture for lower-reward policies but advantages for guided training in the two topmost intervals. Lunar Lander is a Gymnasium (Towers et al. 2024) domain. We generated a pool with 1000 state pairs. The results are very positive, with guided training reducing the number of bug states across all intervals. Dynamic Obstacle is again a Minigrid domain. The pool consists of 500 state pairs. The results are modestly positive, with the mean values for guided training being slightly below those for standard training, consistently across intervals.

Intersection simulates a simplified autonomous driving (Leurent 2018) scenario. We generated a pool with 500 state pairs. The results for guided training are mixed here, with good bug reductions in the intervals from 4.5 to 5 and mild bug reduction in the top interval, but increased bugs in the interval $5 – 5.5$. Bipedal is again from Gymnasium. We generated a pool with 1000 state pairs. Our results for this domain are mixed. Guided training can effectively reduce the number of bug states for policies with rewards ranging from 230 to 250, but for the other intervals the bug distributions are similar for guided vs. standard training. One reason for this may be the continuous action space of Bipedal, in difference to the discrete actions in all other domains here. Highway simulates another simplified autonomous driving scenario (Leurent 2018). We generated a pool with 1000 state pairs. The results here are clearly positive, with substantial advantages for guided training across intervals. Overall, across our 6 domains we get two clearly positive results, two mildly positive results, and two mixed/inconclusive results.

Fig. 2 complements the above with information about the number of test points within each reward interval. This serves the following two purposes:

1) It shows the sample size for the distributions in Fig. 1. Most sample sizes are reasonable.
2) It serves to assess policy performance on initial states.

In general, the more the weight of the bars in a domain is on the right-hand side, the more frequently does the RL process generate high-reward policies. Guided training somewhat reduces the frequency of high-reward policies, but only mildly so, and top-performance policies are always generated.

Discussion We hypothesize that the main factors which affect the performance of our approach are 1) the complexity of domains, 2) the inherent instability of DRL algorithms, 3) the number of bug states found, and 4) the hyperparameters of DRL algorithms. For a hard domain (like Bipedal), DRL algorithms are more likely being unstable. Guided-training is sensitive to unstable training because the policy would quickly forget how to solve the task from the initial states once it is trained on bug states. Hence, it basically keeps going back and forth without actually getting better. It is also a problem if a domain has only few bug states, e.g., Go To Object, as it could cause training overfitting to those bugs. This is also the reason why we use a small α-value for this domain. Higher α-value for domains with few bug states could easily result in overfitting.

Conclusion

We studied how to incorporate policy testing into RL training, with the objective to reduce the number of bug states while maintaining competitive policy performance. Our empirical evaluation across several RL benchmark domains shows that our approach can achieve that objective, though the extent to which that is the case depends on the domain. We believe that the incorporation of policy testing, and other policy analysis results, into (re)training is an important direction that merits further attention, with the ultimate aim of a feedback loop leading to better and better policies.

Acknowledgements

This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science) and by the German Federal Ministry of Education and Research (BMBF) as part of the project MAC-MERLin (Grant Agreement No. 01IW24007).

References

Akazaki, T.; Liu, S.; Yamagata, Y.; Duan, Y.; and Hao, J. 2018. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. In *FM*, 456–465. Springer.

Chevalier-Boisvert, M.; Dai, B.; Towers, M.; Perez-Vicente, R.; Willems, L.; Lahlou, S.; Pal, S.; Castro, P. S.; and Terry, J. K. 2023. Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks. In *NeurIPS*, 73383–73394. Curran Associates.

Corso, A.; Moss, R. J.; Koren, M.; Lee, R.; and Kochenderfer, M. J. 2021. A Survey of Algorithms for Black-Box Safety Validation. *JAIR*, 72: 377–428.

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 *NFM*, 127–142. Springer.

Eisenhut, J.; Schuler, X.; Fiser, D.; Höller, D.; Christakis, M.; and Hoffmann, J. 2024. New Fuzzing Biases for Action Policy Testing. In *ICAPS*, 162–167. AAAI.

Eisenhut, J.; Torralba, Á.; Christakis, M.; and Hoffmann, J. 2023. Automatic Metamorphic Test Oracles for Action-Policy Testing. In *ICAPS*, 109–117. AAAI.

Eniser, H. F.; Gros, T. P.; Wüstholz, V.; Hoffmann, J.; and Christakis, M. 2022. Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing. In *ISSTA*, 52–63. ACM.

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

Gros, T. P.; Groß, J.; Höller, D.; Hoffmann, J.; Klauck, M.; Meerkamp, H.; Müller, N. J.; Schaller, L.; and Wolf, V. 2023. DSMC Evaluation Stages: Fostering Robust and Safe Behavior in Deep Reinforcement Learning - Extended Version. *TOMACS*, 33: 1–28.

Gros, T. P.; Müller, N. J.; Höller, D.; and Wolf, V. 2024. Safe Reinforcement Learning Through Regret and State Restorations in Evaluation Stages. In *PV*, 18–38. Springer.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *ICAPS*, 408–416. AAAI.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *ICAPS*, 422–430. AAAI.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *AAAI*, 8064–8073. AAAI.

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

Leurent, E. 2018. An Environment for Autonomous Driving Decision-Making. https://github.com/eleurent/highway-env.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-Level Control Through Deep Reinforcement Learning. *Nature*, 518: 529–533.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347.

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. P.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529: 484–489.

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: 1140–1144.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022a. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *ICAPS*, 629–637. AAAI.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022b. Learning Generalized Policies without Supervision Using GNNs. In *KR*, 474–483. IJCAI.

Steinmetz, M.; Fiser, D.; Eniser, H. F.; Ferber, P.; Gros, T. P.; 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 *ICAPS*, 353–361. AAAI.

Towers, M.; Kwiatkowski, A.; Terry, J. K.; Balis, J. U.; Cola, G. D.; Deleu, T.; Goulão, M.; Kallinteris, A.; Krimmel, M.; KG, A.; Perez-Vicente, R.; Pierré, A.; Schulhoff, S.; Tai, J. J.; Tan, H.; and Younis, O. G. 2024. Gymnasium: A Standard Interface for Reinforcement Learning Environments. *CoRR*, abs/2407.17032.

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

Wang, R. X.; and Thiébaux, S. 2024. Learning Generalised Policies for Numeric Planning. In *ICAPS*, 633–642. AAAI.