Debugging a Policy: A Framework for Automatic Action Policy Testing

Marcel Steinmetz, Timo P. Gros, Philippe Heim, Daniel Höller, Jörg Hoffmann
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany
{steinmetz, timopgros, hoeller, hoffmann}@cs.uni-saarland.de, s8phheim@stud.uni-saarland.de

Abstract

Neural network (NN) action policies are an attractive option for real-time action decisions in dynamic environments. Yet this requires a high degree of trust in the NN. How to gain such trust? Systematic *testing* certainly is one possible answer, in analogy to program testing. The input to the program becomes the start state for the policy; and erroneous program behaviors – "bugs" – become bad policy behavior, e.g. not reaching the goal from a solvable state. We introduce a framework spelling out this concept. The framework is generic and in principle applicable to arbitrary planning models. We discuss how this form of testing can be operationalized, i.e., how to confirm a bug has been found, and how potential bugs might be identified in the first place. This essentially involves seeing standard planning concepts through the new lense of policy testing. The implementation and practical exploration of this framework remains open for future work. We believe that action policy testing is an important topic for ICAPS, and we hope that our framework will serve to start its discussion.

1 Introduction

Neural networks (NN) are an increasingly important representation of action policies. In basic AI research, huge successes were achieved in game playing (Mnih et al. 2013; Silver et al. 2016, 2018), and an increasing body of work is dedicated to learning action policies in planning (Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020). Within and beyond basic research, a growing trend is to rely on NN action policies for real-time decision taking in dynamic environments. The vision is elegant and simple: a single call to the NN policy suffices to obtain the next action.

Yet this vision comes with high requirements on trust in the neural network. How to gain such trust? There are various possible avenues, including for example any manner of explainable AI that may help to elucidate the NN's action decisions; NN action policy verification, which might leverage progress on verifying NN decision episodes (Katz et al. 2017; Gehr et al. 2018), but will ultimately need to tackle the combined complexity of the state space explosion *and* NN analysis; or shielding, which augments the NN action policy with a safety guard (Könighofer et al. 2017; Alshiekh et al. 2017; Fulton and Platzer 2018). Here we address systematic *testing*, which certainly also is one possible answer, in analogy to program testing. The program is the NN; input to the program is the start state for the policy; erroneous program behavior is bad policy performance; a "bug" is a state where the tested policy achieves much less than could be achieved (e.g., fail to reach the goal even though the state is solvable).

Once such bug-states are identified, they can potentially be used for re-training, thus closing the loop with reinforcement learning. However, we believe that test-case generation is an important research challenge in itself. First, even if viewed as a sub-problem of RL, this is highly non-trivial and must be understood separately. Second, test-case generation is of interest in its own right as a way to *understand* the strengths and weaknesses of the current policy, in terms of (groups of) start states with specific quality issues. Ultimately, the aim is to **certify** through extensive testing that the policy can be trusted. We conjecture that such certification will become increasingly important as the use of neural network policies proliferates.

Test-case generation for NN action policies relates to two major strands of research: test-case generation for software (Clarke 1976; King 1976) and adversarial ML/neural network robustness attacks (e.g. (Carlini and Wagner 2017; Hein and Andriushchenko 2017; Wong and Kolter 2018)). We expect that a wealth of ideas will be suitable for transfer from these areas. Observe, however, that these will need to be adapted in non-trivial ways. Test-case generation for software is obviously different in terms of code vs. a neural network; furthermore, action choice in sequential decision making can be viewed as a special case of software, but comes with manifold important particularities including e.g. exponential state spaces and uncertain environments. As for adversarial attacks, these are concerned with single classifier decision episodes, whereas here we are concerned with sequences of classifier decision episodes in a large state space. While classifier deficiencies become immediately apparent in the former, a bad decision in the latter may exhibit its adverse consequences only much later on.

In the present paper, we introduce a framework for action-policy testing. The framework is generic and in principle applicable to arbitrary planning models, from classical planning to POMDPs; we illustrate the framework with a range of example models. The basic idea of our framework, as al-

ready hinted above, is quite simple: A bug is a state on which the learned NN action policy achieves (much) worse performance than an optimal policy. The practical question arising immediately from this of course is, how can we **confirm** we found a bug in practice, where the optimal policy will not be known (or else there would be no point in learning a policy in the first place)? What could be practical **bug-candidate generation** methods given these observations?

We address these questions in turn, discussing first planning models and NN action policies, defining what bugs are, then outlining the basic observations regarding bug confirmation, and fuzzing as a method to generate test cases. All this essentially involves seeing standard planning concepts through the new lense of policy testing. The implementation and practical exploration of this framework remains open for future work. We believe that action policy testing is an important topic and should be present at ICAPS, and we hope that our framework will serve to start its discussion.

2 Planning Models

Traditionally, action policies – functions from states to actions, tackling the entire state space or at least a large fraction thereof – are being considered only in probabilistic planning models (e.g. (Younes et al. 2005; Sanner 2010; Coles et al. 2012)), as in other models more compact plan representations exist. The most striking example is deterministic planning (e.g. (McDermott et al. 1998; Bacchus 2000; Fox and Long 2003; Hoffmann and Edelkamp 2005; Gerevini et al. 2009)), where a plan is simply an action sequence; in contingent planning (e.g. (Peot and Smith 1992; Hoffmann and Brafman 2005; Muise, Belle, and McIlraith 2014)), action-tree representations become large in the worst case but can be small if only few observation (sensing) actions are required. However, also in those cases action-policy learning can make sense, for the sake of generalization. This is commonly acknowledged in per-domain generalization, learning policies applicable to all instances of a domain (e.g. (Fern, Yoon, and Givan 2006; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020)); generalization over goals is weaker but also of obvious interest. Recent work (Ferber, Hoffmann, and Helmert 2020) argues that even generalization over only the states in a fixed deterministic state space (fixed universe, fixed goal, fixed initial state) makes sense, in applications where one expects to be confronted with arbitrary states during online planning (e.g. due to environment noise).

Regardless whether or not one accepts the latter argument, it certainly makes sense to not make unnecessary restrictions on the planning model considered. The framework we define in what follows is generic and (potentially) applicable in arbitrary contexts. We merely assume that one can define a function V^* assigning states s to the optimal value any policy can achieve on s; the algorithmic methods we outline are based on approximations of V^* as well as of the value a given policy π achieves on s. As examples covering a broad range of planning-model differences, we will consider:

- **Classical planning**. The initial state is unique (complete knowledge), action outcomes are deterministic. The objective is to reach a goal condition.

  Optionally, a cost function assigns each action a non-negative cost and the objective is to minimize summed-up cost where an action sequence not reaching the goal has cost ∞.
- **Contingent planning**. There is a set of possible initial states called the initial belief. Actions can have several possible outcomes, one of which occurs non-deterministically. Sensing actions for a subset of state variables allow to observe the current value of such a variable. The objective is to reach a goal condition with certainty, i.e., guarantee to reach a belief state in which all states satisfy the goal.

  Optionally the objective is to minimize worst-case cost.

  The policy in this case maps belief states (not states) to actions.[1] Contrary to convention, we will denote belief states simply with s like other states, to avoid having to make case distinctions irrelevant to our framework.
- **Oversubscription planning**. Like classical planning, but with a bound on plan cost, and with a set of goal conditions each of which is associated with a positive reward. The objective is to maximize summed-up reward.
- **Discounted-reward MDPs**. No constraints on the initial state. State transitions are probabilistic due to uncertain action outcomes, or environment noise, or both. A reward function maps states or state-action pairs to reals. The objective is to maximize (infinite-horizon) expected discounted reward.
- **MaxProb MDPs**. Infinite-horizon non-discounted MDPs with the objective to maximize the probability to reach the goal.

One can furthermore distinguish whether the policy is deterministic (every state is mapped to a unique action) or non-deterministic (several actions are possible, one of which is chosen arbitrarily or according to a probability distribution). In our example models, non-deterministic policies are not required to obtain optimal solutions, so we will consider only deterministic policies in what follows.

3 What is a "Bug"?

We next introduce and discuss our definition of bugs.

3.1 Definitions

As indicated, our only assumption is that one can define a **value function** $V : S \times \Pi \mapsto \mathbb{R}$ which maps state/policy pairs (s, π) to numbers capturing the performance of π on s. We will denote $V^\pi(s) := V(s, \pi)$ in line with convention.

In quantitative settings, V is simply defined by the optimization objective, namely (worst-case) summed-up cost, (expected) summed-up reward, and goal probability in our example models. In qualitative settings, where the objective

[1] In practice, the actual input to the policy are observation histories, from which the belief state must first be inferred. This does not make a difference for our purposes here, and considering the policy to be a function on belief states is conceptually simpler for our discussion.

is to reach a goal condition, the following definition of V often makes sense:

Definition 1 (Qualitative Value Function)**.**

$$V^\pi(s) := \begin{cases} 0 & \text{no run of } \pi \text{ on } s \text{ reaches the goal} \\ 0.5 & \text{some runs of } \pi \text{ on } s \text{ reach the goal} \\ 1 & \text{all runs of } \pi \text{ on } s \text{ reach the goal} \end{cases}$$

where the objective is maximization.

These notations accumulate standard concepts under the notation of a value function. Along the same lines, we denote the value of the optimal policy by $V^* : S \mapsto \mathbb{R}$ where[2]

Definition 2 (Optimal Value Function)**.**

$$V^*(s) := \begin{cases} \min_\pi V^\pi(s) & \text{objective is minimization} \\ \max_\pi V^\pi(s) & \text{objective is maximization} \end{cases}$$

Now a bug is very easy to define in general terms:

Definition 3 (Bug)**.** A state s is a **bug** in policy π if $|V^\pi(s) - V^*(s)| > 0$. We also say that s is a **Δ-bug** where $\Delta := |V^\pi(s) - V^*(s)|$; and that s is a **normalized Δ-bug** if $V^*(s) \neq 0$ and $\Delta := |\frac{V^\pi(s) - V^*(s)}{V^*(s)}| > 0$.

For example, in classical or contingent planning minimizing cost, a normalized 0.5-bug is a state on which π incurs cost 50% higher than optimal. Similarly, in oversubscription planning or discounted-reward MDPs, a normalized 0.75-bug is a state on which π attains only 25% of the possible reward. In MaxProb MDPs, a 0.5-bug is a state on which π reaches the goal with 0.5 less probability than possible. Note here that normalized Δ-bugs make more sense in the first two cases where the range of costs/rewards differs across instances; whereas probabilities are already normalized.

In qualitative settings, Definition 3 takes on a very specific form according to Definition 1 which merely distinguishes three possible cases. In classical planning, either every policy run from s reaches the goal, or none does; so the only possible bug is a 1-bug which simply means that s is solvable but π does not reach the goal. This is at the same time the simplest and intuitively strongest form of a bug in our framework. We will refer to it as **fail-bug**. In contingent planning, all three cases are possible, i.e., the belief states the policy can reach (given different sensing outcomes) may only partially ascertain the goal. A 0.5-bug means that the policy fails to reach the goal in some cases, which we refer to as **somefail-bug**; a 1-bug means that the policy fails to reach the goal in all cases, which we again call a **fail-bug**. Of course, in a setting where the policy may reach the goal in some but not all cases, the fraction of solved cases provides a more fine-grained quantitative value function.

3.2 Discussion

We say that a policy π is **bug-free** if there are no bugs. This is obviously the case iff π is optimal on all states. In many settings, in particular when information about the initial state is given (a unique initial state or an initial belief), it makes sense to restrict this notion to states that are reachable from at least one possible initial state.

Another natural notion of "bug" could be any state s where $\pi(s)$ does not start any optimal policy for s (or is even inapplicable). This is a special case of our definition: any such s is a bug, but not vice versa; and a policy free of only such bugs is not necessarily optimal. This is because it is not enough to follow some optimal policy in each state. These choices need to be coordinated across states. As a simple illustration, consider the state space $g_1 \overset{b}{\leftarrow} s_1 \overset{a}{\leftrightarrow} s_2 \overset{b}{\rightarrow} g_2$ where the objective is to reach the goal condition $g_1 \vee g_2$ (qualitative, no cost minimization). Consider any policy where $\pi(s_1) = \pi(s_2) = a$. Such policies do follow some optimal policy in each of s_1 and s_2, yet never reach the goal. Such a notion of bug can, therefore, at most be employed as a sufficient criterion to find bugs in practice. We consider this and more general methods next.

Some words are finally in order on the expected relation between the (learned) action policy π vs. the planning model and optimization objective. In the canonical scenario, (a) π has been learned on the same planning model it is tested against, and (b) the learning objective for π has been the same as the optimization objective it is tested against. But there are reasonable scenarios in which either or both of these conditions is false. Regarding (a), π may be learned externally to the planning model, in which case that model plays the traditional role of models in model checking, formalizing π's environment so as to be able to check π's properties.[3] Regarding (b), it is conceivable that the testing process may focus on different criteria. For example, while a policy might be learned to maximize discounted reward, testing might be used to certify the absence of failures (like crashes), i.e., using the qualitative value function requiring to reach the goal wherever possible. Such a scenario makes sense, for example, if learning a policy for combined criteria (like maximizing reward subject to a limit on crash probability) is not feasible; or as part of a broad testing process certifying the policy against an entire range of criteria.

4 How to Confirm we Found a Bug?

Let's assume that we identified a state s by some (yet undefined) bug-candidate generation method, and now want to confirm whether or not s is indeed a bug. This is difficult in practice in case we don't know $V^*(s)$. One might expect that this will always be the case, because otherwise learning a policy would be pointless. There are however relevant exceptions to this rule. We next discuss these, then turn to the general case.

4.1 Easy Cases

In the following scenarios, it is feasible to compare $V^\pi(s)$ against $V^*(s)$ at least on some states:

[2] In case the state space, and therewith the number of policies, is infinite, a minimum/maximum may not exist; model-specific notions of optimality apply then. For our purposes here, the only thing that counts is that an optimal value function can be defined.

[3] Even in this scenario though π must be interpretable as a policy for the model, so adapters must be defined between the model and the policy input (states) and output (actions).

- We may require a learned policy *online* to take decisions in real time, while *offline* (in particular during testing) it is feasible to compute $V^*(s)$. (Scientific experiments evaluating bug-candidate generation methods may simply assume that this is the case.)
- A realistic testing scenario are sanity tests, with policy behavior being tested on simple regions of the state space, where the user – the engineer testing the action policy – knows the value of V^*, or the optimal action choices, by design. For example, the user could supply a formula ϕ in contingent planning, characterizing belief states from which we know the goal can be reached with certainty, $V^*(s) = 1$ for $s \models \phi$. Another very simple example is the sanity test checking whether the action returned by the policy is applicable (which makes sense only if the design does not enforce this, e.g. by restricting a final softmax layer to consider only applicable actions).
- We can in some settings design our testing machinery so as to generate only state s with a given $V^*(s)$ value. This pertains primarily (probably exclusively) to qualitative settings where we know a plan exists. For example, this is the case if we generate s by backward steps from the goal.
- In some qualitative settings, we can derive $V^*(s)$ from easy structural analyses. For example, in classical planning we have qualitative $V^*(s) = 1$ globally if the task is known to be solvable, and all actions are invertible or at least undoable (Daum et al. 2016).
- Sometimes, qualitative $V^*(s)$ can be determined by a fixed-depth lookahead search. For example, when navigating traffic, escaping a dangerous situation (breaking/changing the lane/stopping at the right-hand side) is a local activity whose required lookahead depth does not depend on the length of the planned trajectory.

Note that all but the first scenario pertain to qualitative value, i.e., plan existence. For numeric value optimization, special cases with known optimal value are sparse. Hence it is certainly important to address the general case.

4.2 The General Case: Approximations

A natural approach is to turn to approximations, deriving lower bounds on the value gap $\Delta = |V^\pi(s) - V^*(s)|$, i.e., lower bounds $L \leq \Delta$. If $L > 0$ then we know that s is a bug. Such bounds L can indeed be derived; they boil down though to finding a plan for s that achieves quality better than π.

To confirm this observation in general terms, and also for the discussion of fuzzing-bug confirmation in the next section, we need a generic "better than" notation that we can use across minimization vs. maximization of state values. For any value function V, we denote $V(s') \prec V(s)$:iff

$$\begin{cases} V(s') < V(s) & \text{objective is minimization} \\ V(s') > V(s) & \text{objective is maximization} \end{cases}$$

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

Lower bounds $L(s)$ can be derived by a combination of optimistic and pessimistic approximation. Namely, say that H_* is a pessimistic approximation of the optimal value, i.e. $V^*(s) \preceq H_*(s)$; and h_π is an optimistic approximation of π's value, i.e. $h_\pi(s) \preceq V^\pi(s)$. Under additional sanity conditions on these two approximations, we get the desired lower bound:

Proposition 4 (Bug Confirmation)**.** Say that $V^*(s) \preceq H_*(s)$ and $h_\pi(s) \preceq V^\pi(s)$. If, in addition, (i) $h_\pi(s) \succeq V^*(s)$ and (ii) $H_*(s) \preceq V^\pi(s)$, then $|h_\pi(s) - H_*(s)| \leq |V^\pi(s) - V^*(s)|$.

Proof. Under the stated conditions, we have $V^*(s) \preceq H_*(s) \preceq V^\pi(s)$ and $V^*(s) \preceq h_\pi(s) \preceq V^\pi(s)$, i.e., both $H_*(s)$ and $h_\pi(s)$ lie in between $V^*(s)$ and $V^\pi(s)$. □

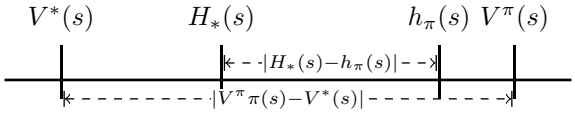

Figure 1: Illustration of Proposition 4.

Intuitively, the sanity conditions (i) and (ii) prevent the approximations from being "too bad": (i) says that we cannot approximate the policy value $V^\pi(s)$ to be better than optimal; (ii) says that we cannot approximate the optimal value $V^*(s)$ to be worse than that provided by the policy itself. Note that these conditions are indeed required. If $h_\pi(s)$ is better than $V^*(s)$, or $H_*(s)$ is worse than $V^\pi(s)$, then we can widen the distance between the two arbitrarily far beyond $|V^\pi(s) - V^*(s)|$.

Satisfying (i) may sound difficult at first, but is actually natural as $V^*(s) \preceq V^\pi(s)$ and policy values can be easily approximated statistically (in deterministic cases, a single run of the policy yields its exact value). As for (ii), we can theoretically always ensure this by taking the policy value $V^\pi(s)$ as the initial pessimistic approximation. That value itself will however never lead to a bug identification, as we are then using the same value for both $h_\pi(s)$ and $H_*(s)$. We need to find a pessimistic approximation where $H_*(s) \prec V^\pi(s)$.

Hence, overall, the bug confirmation method arising from the above simply is to (i) approximate $V^\pi(s)$ up to a high degree of precision, and (ii) try to find a better policy for s. It may seem disappointing that nothing deeper can be done here. But certainly this method can be quite practical. In classical planning, (ii) can be instantiated with fast greedy satisficing planning methods that scale much better than optimal planning (e.g. (Hoffmann and Nebel 2001; Gerevini, Saetti, and Serina 2003; Richter and Westphal 2010; Rintanen 2012; Domshlak, Hoffmann, and Katz 2015)), trying to find plans that cost less than the plan found by π, or that reach the goal in case π does not. A particularly interesting option also are bounded-suboptimal searches, and searches pruning against an initial upper bound, which in our context become decision procedures checking whether or not a plan better than π exists. Even more targeted than this, there is previous work aiming at solving precisely the

sub-problem we are facing here: posthoc plan-quality optimization, which is geared at improving the quality of an existing plan (e.g. (Bäckström 1998; Do and Kambhampati 2003; Nakhost and Müller 2010)). These can be plugged in for the purpose of bug confirmation. Beyond classical planning, similar methods work in principle, though the arsenals of prior techniques that can be plugged in are not as large. Anytime methods starting from π and trying to improve it could make sense in most contexts.

5 Bug-Candidate Generation via Fuzzing

Bug-candidate generation methods are needed to identify interesting states s, i.e., potential bugs, in the first place. Fuzzing is a classical approach in test-case generation, starting from some input and applying random perturbations, ideally with a bias that may favor the generation of bugs. In what follows, we outline the basic aspects of this idea in our framework. The natural target of fuzzing here is to widen the optimality gap: move from a given state s to a fuzzed state s' on which the policy behaves worse.

In what follows, we first spell out this idea. We then discuss how potential bugs s' found this way can be confirmed. We clarify how this relates to the bug-confirmation method from the previous section. We finally discuss practical matters in the design of fuzzing methods in our context.

5.1 Fuzzing Bugs

Say we are given a state s (which itself might have been generated randomly in one or the other way, see Section 5.4). We perform randomized perturbations to obtain another state s'. In our context, this operation can be considered a success if it widens the optimality gap:

Definition 5 (Fuzzing Bug)**.** A state s' is a **fuzzing-bug** relative to s if $|V^\pi(s') - V^*(s')| > |V^\pi(s) - V^*(s)|$. We also say that s is a Δ**-fuzzing-bug** where $\Delta := |V^\pi(s') - V^*(s')| - |V^\pi(s) - V^*(s)|$.

Observation 6 (Fuzzing Bugs are Bugs)**.** If s' is a fuzzing-bug relative to some s, then s' is a bug.

Proof. The worst-case optimality gap $|V^\pi(s') - V^*(s')|$ of s' arises when s is not a bug, $|V^\pi(s) - V^*(s)| = 0$. In this case $|V^\pi(s') - V^*(s')| = \Delta$ when s' is a Δ-fuzzing-bug relative to s. □

A fuzzing-bug is a state we can show to be a bug through fuzzing, widening the optimality gap with respect to a starting state. One may wonder whether that concept is actually restricting, i.e., whether the class of fuzzing-bugs is small relative to that of bugs. In general, this is not so:

Observation 7 (Bugs are (almost) Fuzzing Bugs)**.** Every bug s' with non-minimal optimality gap $|V^\pi(s) - V^*(s)|$ is a fuzzing-bug relative to some s.

Proof. Any state s with minimal optimality gap satisfies the claim. □

Hence, absent algorithmic restrictions on what "we can show to be a bug through fuzzing", essentially every bug is a fuzzing bug. In particular, as soon as there is at least one non-bug state – whose optimality gap is 0 – all bugs are fuzzing-bugs. In some planning models, this is even guaranteed by construction. For instance, goal states are necessarily bug-free in all policies. The same applies to terminal states, and states with $V^*(s) = 0$ when there are no negative rewards.

Given all this, one may wonder about the point of having a concept "fuzzing-bug". However, in actual fuzzing algorithms there will be restrictions on the relation between s and s', such as reachability, or reachability in a limited number of fuzzing operations; and there may be restrictions on the states s to start from. In this setting the notion of fuzzing-bug is more restricting. More prosaically, the fuzzing-bug concept is useful for the following technical discussion, as it sets the precise context that discussion pertains to.

5.2 Fuzzing Bug Confirmation

To make effective use of fuzzing-bugs, we need to somehow test whether the optimality gap from s to s' has widened without relying on the precise gap values. Observe that Definition 5 encompasses two possible cases: (a) s' is better (or at least as good as) s in terms of optimal value, but π performs worse at s' or at least does not fully pick up this advantage; (b) s' is worse than s in terms of optimal value, but the disadvantage in π is even worse:

Proposition 8 (Ideal Fuzzing Bug Confirmation)**.** s' is a fuzzing-bug relative to s iff one of the following two conditions holds:

(a) $V^*(s') \prec V^*(s)$, and either $V^\pi(s') \succeq V^\pi(s)$ or $|V^\pi(s') - V^\pi(s)| < |V^*(s') - V^*(s)|$;

(b) $V^*(s') \succeq V^*(s)$, $V^\pi(s') \succeq V^\pi(s)$, and $|V^\pi(s') - V^\pi(s)| > |V^*(s') - V^*(s)|$.

Proof. "If" direction: In case (a), optimal value gets better from s to s' but policy value does not, or not as much. In case (b), both values can only get worse but the extent of that change is larger for V^π. "Only if" direction: clearly if $|V^\pi(s') - V^*(s')| > |V^\pi(s) - V^*(s)|$ then one of these two cases must be satisfied. □

These are ideal conditions in the sense that they capture fuzzing-bugs exactly. But they cannot be checked directly. V^π, as discussed above, is relatively easy to measure by executing the policy (several times in case of nondeterministic/probabilistic behavior). Both cases however also require information on whether the optimal value got better or worse, and on the exact extent of that change. In what follows, we discuss in how far bounds on V^* can be leveraged to obtain sufficient conditions. Specifically, we discuss the use of pessimistic bounds $H_* \preceq V^*$ and optimistic bounds $h_* \preceq V^*$, evaluated on both s and s'. Both kinds of bounds are available or can be designed, in many planning models. Pessimistic bounds have already been discussed above; optimistic bounds are the standard notion of admissible heuristic functions, for which the planning literature offers many possibilities, in classical planning and beyond (e.g. (Haslum and Geffner 2000; Edelkamp 2001; Helmert and Domshlak 2009; Helmert et al. 2014; Domshlak and Mirkis 2015;

Trevizan, Thiébaux, and Haslum 2017; Seipp, Keller, and Helmert 2020; Klößner et al. 2021)).

We next identify two fuzzing-bug confirmation methods based on such bounds. Our first method applies to a scenario where we can lower bound the change in optimal value. Consider the two intervals $I_*(s) := \{v \mid h_*(s) \preceq v \preceq H_*(s)\}$ and $I_*(s') := \{v \mid h_*(s') \preceq v \preceq H_*(s')\}$ limiting the range of possible optimal values at s and s' respectively. Observe that, if $I_*(s)$ and $I_*(s')$ are disjoint $I_*(s) \cap I_*(s') = \emptyset$, then we know the direction of the V^* change from s to s', and hence which case (a) or (b) of Proposition 8 we have to check. In particular:

Proposition 9 (Fuzzing Bug Confirmation (a))**.** Assume that $I_*(s) \cap I_*(s') = \emptyset$. Then, s' is a fuzzing-bug relative to s if $H_*(s') \prec h_*(s)$ and either $V^\pi(s') \succeq V^\pi(s)$ or $|V^\pi(s') - V^\pi(s)| < |H_*(s') - h_*(s)|$.

Proof. This situation is depicted in Figure 2. With $H_*(s') \prec h_*(s)$, we have $V^*(s') \prec V^*(s)$. Moreover, with $I_*(s) \cap I_*(s') = \emptyset$ and $H_*(s') \prec h_*(s)$, $|H_*(s') - h_*(s)|$ is exactly the minimum distance between the two intervals, $|H_*(s') - h_*(s)| = L_*(s, s')$ where $L_*(s, s') := \min\{|v - v'| \mid v \in I_*(s), v' \in I_*(s')\}$. Hence $|H_*(s') - h_*(s)| \leq |V^*(s') - V^*(s)|$. The claim follows from Proposition 8 (a). □

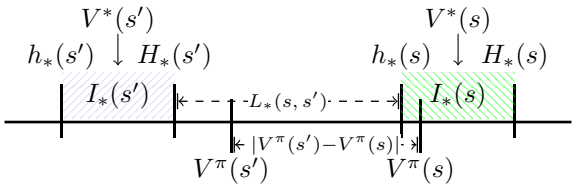

Figure 2: Illustration of the relationship between the bounds in Proposition 9.

Given the bounds H_* and h_*, we can easily derive $I_*(s)$ and $I_*(s')$, check whether they intersect, compute $L_*(s, s')$ if that is so, and test the prerequisites of Proposition 9. So this gives us a practical method for fuzzing-bug confirmation. That method will be of limited practical value though: presumably, given the information loss usually incurred in particular by optimistic bounds, it will be rare that $I_*(s)$ and $I_*(s')$ are disjoint.

Our second fuzzing-bug confirmation method relies on an upper bound on the V^* change instead. Its key advantage is that, while Proposition 9 is limited to the case where V^* got better from s to s', the new method applies regardless of the direction of the change.

Proposition 10 (Fuzzing Bug Confirmation (b))**.** Let $U_*(s, s') \geq |V^*(s') - V^*(s)|$ be an upper bound on the change in optimal value between s and s'. Then, s' is a fuzzing-bug relative to s if $V^\pi(s') \succeq V^\pi(s)$ and $|V^\pi(s') - V^\pi(s)| > U_*(s, s')$.

Proof. If $V^*(s') \succeq V^*(s)$, the prerequisites clearly imply Proposition 8 (b). If $V^*(s') \prec V^*(s)$, then $V^\pi(s') \succeq V^\pi(s)$ implies Proposition 8 (a). □

This is a worst-case analysis in the sense that the stated condition still works under the pessimistic assumption that V^* got worse, (b), as in this case V^π must have deteriorated by an even larger amount. In addition, we can also obtain the required upper bound $U_*(s, s')$ regardless of the direction of the V^* change:

Proposition 11 (V^* Change Upper Bound)**.** Let $\delta_1 := |H_*(s') - h_*(s)|$, $\delta_2 := |h_*(s') - H_*(s)|$, and $U_*(s, s') := \max(\delta_1, \delta_2)$. Then $|V^*(s') - V^*(s)| \leq U_*(s, s')$.

Proof. If $V^*(s') \preceq V^*(s)$ then $|V^*(s') - V^*(s)| \leq \delta_2$; otherwise $|V^*(s') - V^*(s)| \leq \delta_1$. □

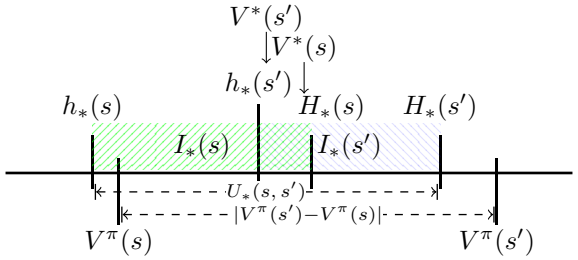

Figure 3: Illustration of Proposition 10 and Proposition 11.

Figure 3 shows how this result integrates into Proposition 10. As illustrated, even if the measured intervals $I_*(s)$ and $I_*(s')$ overlap, and hence we have no information on whether $V^*(s)$ or $V^*(s')$ is better, s' can nevertheless be identified as a fuzzing-bug.

We arrive at the following fuzzing-bug confirmation method: measure $V^\pi(s')$ and $V^\pi(s)$; run optimistic and pessimistic approximations of $V^*(s')$ and $V^*(s)$, and compute $U_*(s, s')$; if $V^\pi(s') \succ V^\pi(s)$ and $|V^\pi(s') - V^\pi(s)| > U_*(s, s')$, then s' is a fuzzing-bug relative to s.

5.3 Comparison of Bug Confirmation Methods

We now have three practical bug confirmation methods in hand: (1) Proposition 4 which confirms bugs s' by finding a better policy at s' (pessimistically approximating V^*); (2) the method based on Proposition 9, which exploits disjoint intervals $I_*(s)$ and $I_*(s')$; and (3) the method just sketched, based on Proposition 10. How do these methods relate to each other?

Methods (2) and (3) differ in covering the different cases of Proposition 8. They are complementary in that (3) but not (2) applies when $I_*(s)$ and $I_*(s')$ overlap; while (2) but not (3) may cover cases where the policy value got better $V^\pi(s') \prec V^\pi(s)$.

But what is the relation of (2) and (3) to (1)? In particular, can the more complicated fuzzing-bug conditions identify bugs that cannot be identified by (1)? Unfortunately, and somewhat surprisingly, the answer to that question is "no":

Theorem 12. Let s and s' be arbitrary states. Consider the same bounds $h_* \preceq V^*$ and $H_* \succeq V^*$ in all methods. If Proposition 9 or Proposition 10 confirm s' to be a bug, then so does Proposition 4.

Proof. We show that Proposition 9 and Proposition 10 each imply $H_*(s') \prec V^\pi(s')$. Therefore, if they are satisfied, then s' also satisfies Proposition 4 for the considered pessimistic bound. Assume for contradiction that $H_*(s') \succeq V^\pi(s')$.

If Proposition 9 is satisfied, its prerequisite connects $V^\pi(s)$ and $V^\pi(s')$ through the following chain of relations: $V^\pi(s) \succeq h_*(s)$ because h_* is optimistic; $h_*(s) \succ H_*(s')$ per prerequisite of Proposition 9; $H_*(s') \succeq V^\pi(s')$ per our assumption. Overall, we get $V^\pi(s) \succeq h_*(s) \succ H_*(s') \succeq V^\pi(s')$ and hence $|V^\pi(s') - V^\pi(s)| \geq |H_*(s') - h_*(s)|$. Now, as argued in the proof of Proposition 9, the minimum distance $L_*(s, s')$ between the intervals $I_*(s)$ and $I_*(s')$ is exactly $|H_*(s') - h_*(s)|$. Put together, we get $|V^\pi(s') - V^\pi(s)| \geq L_*(s, s')$, which is a contradiction to the prerequisite of Proposition 9.

Next consider Proposition 10. By assumption, $H_*(s') \succeq V^\pi(s')$; the first condition of Proposition 10 gives $V^\pi(s') \succeq V^\pi(s)$; since $U_*(s, s')$ is non-negative by definition, the second condition entails that $|V^\pi(s') - V^\pi(s)| > 0$, so the inequality in our previous observation is actually strict ($V^\pi(s') \succ V^\pi(s)$); and finally $V^\pi(s) \succeq h_*(s)$ since h_* optimistically bounds V^*. We arrive at the following chain of relations: $H_*(s') \succeq V^\pi(s') \succ V^\pi(s) \succeq h_*(s)$. Therefore, $|V^\pi(s') - V^\pi(s)| \leq |H_*(s') - h_*(s)|$. Since $|H_*(s') - h_*(s)|$ is exactly δ_1 from Proposition 11 and $\delta_1 \leq U_*(s, s')$, we conclude that $|V^\pi(s') - V^\pi(s)| \leq U_*(s, s')$. This contradicts the condition of Proposition 10. □

To understand this result intuitively, consider the following. Proposition 9 requires the intervals $I_*(s)$ and $I_*(s')$ to be disjoint, with $I_*(s')$ being strictly better. As shown in Figure 2, if the objective is minimization, this means that $I_*(s') = [h_*(s'), H_*(s')]$ lies below $I_*(s) = [h_*(s), H_*(s)]$. At the same time, $V^\pi(s)$ lies above $h_*(s)$ and $V^\pi(s')$ needs to be close to $V^\pi(s)$. Hence $H_*(s')$ naturally lies below $V^\pi(s')$. For Proposition 10 the situation is symmetric. Consider Figure 3. The intervals $[h_*(s'), H_*(s')]$ and $[h_*(s), H_*(s)]$ must remain close together, while $V^\pi(s')$ must become worse than $V^\pi(s)$. With $V^\pi(s)$ being close to $[h_*(s), H_*(s)]$ and thus to $[h_*(s'), H_*(s')]$, again $H_*(s')$ naturally lies below $V^\pi(s')$. Theorem 12 shows that these intuitions are actually necessities, with $H_*(s')$ necessarily below $V^\pi(s')$, enabling bug confirmation via Proposition 4.

While this is a negative result, note that fuzzing-bug confirmation may still make sense in practice. All the theorem says is that one *can* also confirm the bug via method (1). This may be more difficult in practice however. For example, method (3) can sometimes be used without relying on any (optimistic/pessimistic) bounds, namely when an upper bound on $|V^*(s') - V^*(s)|$ can be derived by simpler means. For example, in classical planning with cost minimization, say that all actions are invertible. If we obtain s' from s by applying actions whose summed-up cost is B, then $|V^*(s') - V^*(s)| \leq B$. In this situation, according to Theorem 12 one could confirm s' to be a bug through satisficing planning finding a plan through s. But obviously this is more expensive (and may not happen, depending on the satisficing planning process).

5.4 Fuzzing Methods

With these theoretical considerations in mind, let us now finally discuss actual fuzzing methods. The canonical view of these is as suitable forms of local search iteratively perturbing a start state – including simple methods like hill-climbing, but potentially also more involved methods such as evolutionary algorithms. In any such setup, key design decisions are:

- **Fuzzing bias**: The biases in our randomized perturbations of s.
- **Fuzzing operators**: The state-modification operators applied in these perturbations.
- **Fuzzing start states**: The set of states s from which we start these operations.
- **Fuzzing termination**: When do we stop?

Fuzzing bias. Natural fuzzing biases arise directly from the bug confirmation methods (2) and (3) above. A fuzzing algorithm relying on (2) needs to make V^* better (finding an easier state s' starting from s) while limiting the change in policy value. One way to achieve this could be heuristic search to obtain a bias towards better V^*, while restricting fuzzing operators to discard state changes that improve V^π. A fuzzing algorithm relying on (3) needs to achieve $V^\pi(s') \succeq V^\pi(s)$ with large $|V^\pi(s') - V^\pi(s)|$, and needs the bound $U_*(s, s')$ from Proposition 11 to be small. The former entails a bias towards states t where π performs badly, and the latter entails a bias to states t where the approximations $H_*(t)$ and $h_*(s)$, as well as $h_*(t)$ and $H_*(s)$, are close to each other. In other words we should try to deteriorate the policy while keeping the optimal value fixed so far as our approximations allow us to.

An important special case in this context are qualitative value functions, where the planning objective is to reach a goal, and the fuzzing objective is to find a state from which the policy does not (or not always, in non-deterministic cases like contingent planning) reach the goal. Note that (2) does not make sense in this case as we will naturally start from s where the policy does reach the goal, so we will need to deteriorate the policy behavior along the lines of (3). The issue is that the value function is characterized by jumps in its surface, not providing a suitable gradient for fuzzing biases. In non-deterministic settings like contingent planning, one can use the fraction of solved cases as a more quantitative value function providing the desired gradient. In deterministic cases however, we are confronted with a value function that remains 1 until we found a bug where it jumps to 0. We hence need measures of "solving difficulty" for the behavior of π on any intermediate state t during fuzzing. Plan/policy-run length (while keeping optimal plan length fixed as per (3)) is one option, based on the assumption that overlong plans indicate a difficulty to reach the goal. A more specific measure may be plan redundancy, i.e., execution traces from which action subsequences can be removed while still reaching the goal. To identify other possible measures, an empirical investigation seems needed into the neighborhoods of states that policies (maybe particular classes of policies,

such as ones based on a particular neural network architecture) fail to solve.

A completely different approach is to ignore bug confirmation during fuzzing, and go for *coverage* instead. This is attractive in particular (but not only) in scenarios where we cannot easily get useful fuzzing biases from approximations as outlined above.

We maintain a pool of test cases $P = \{s_1, \ldots, s_k\}$, and the target of fuzzing is to modify or extend that pool to increase its coverage, defined based on some measure of how "different" the s_i are. In program testing, such measures are defined based on activating different parts of the program. In action policy testing, there are two things we may want to cover: (a) *policy structure*, corresponding to the "program"; and (b) *states*, corresponding to different environment situations the policy may have to deal with. If the policy π has a symbolic representation (e.g.: a program) then (a) is very close to software coverage. If π is a neural network, which is our primary interest here, then (a) can consider neuron activations; to make this meaningful, presumably one needs to identify and distinguish relevant activation patterns or paths. Option (b) is, by comparison, quite straightforward. A baseline for measuring how different states are is simply Hamming distance or variants thereof, comparing state-variable value vectors. A more targeted concept may be novelty (Lipovetzky and Geffner 2012, 2017), the size of the smallest state-attribute combination not seen so far (here: not contained in the pool yet).

Fuzzing operators. Opposing options are to follow the actual planning semantics, applying actions; vs. directly perturbing state-variable values. Actually four different options can be distinguished: (a) apply actions forward, (b) apply actions backward, (c) modify state-variable values, (d) do so but take mutex information into account. An advantage of (a) and (b) is that they can be applied even in situations were no explicit planning model is given and we instead only have access to a state-transition generator (a simulator). Furthermore, with (a), in planning models where information about the initial state is available, starting from reachable s (see (3) below) only reachable fuzzing-bugs are generated. On the other hand, (c) traverses the state space more quickly and may be able to find bugs more effectively; (d) may go some way towards alleviating the reachability concern.

Interesting variants of (c) include e.g. hill-climbing search environments picking some variable and changing its value; but also evolutionary algorithms performing cross-overs between different states in a current candidate set. The latter may be especially suited to coverage-based fuzzing, modifying the pool P by crossing over between the test cases s_i.

Fuzzing start states. Note first that, in program testing, the state/input s started from is not a bug, i.e., program behavior on s is correct and we're trying to move towards a state/input where program behavior is incorrect. In our context, we will not in general directly know whether s itself already is a bug. But one can start fuzzing from s in any case (Proposition 6 holds regardless of whether or not s is a bug). So in principle any starting state s is fine. Like for fuzzing operators, the primary parameter here seems to be whether the states s are generated through action applications, or directly through randomized value assignments to the state variables. This space of options corresponds exactly to different methods for generating random states in planning, resulting in a very similar categorization into four cases: (a) random walks forward from the initial state, or (b) backwards from the goal; (c) random value choice for state variables, optionally (d) taking mutexes into account. With (a) we will generate only reachable fuzzing-bugs; (c) is more flexible and may cover the state space more broadly, with (d) somewhat alleviating the reachability concern.

Option (b) will, combined with the same option for fuzzing operations, limit the fuzzing-bugs found to ones from which the goal can be reached. This is of interest, for example, in deterministic planning models, where, as previously discussed, it implies that the value of the qualitative value function is $V^*(s) = 1$ and thus bug confirmation with respect to that value function is easy.

Fuzzing termination. For fuzzing biases orientated towards bug confirmation, the obvious idea is to stop once we were able to confirm we found a bug. Randomized restarts probably make sense in case of failure, where "failure" could easily be defined based on simple progress criteria regarding the observed changes (or lack thereof) in policy value and/or bounds on the optimal value.

For fuzzing biases based on coverage, it presumably makes sense to define a threshold for when an individual new test case s_{k+1} is "new enough", and/or for when the overall pool $P = \{s_1, \ldots, s_k\}$ has "high enough coverage".

It might make sense to iterate fuzzing processes, rather than restarting them: use the previous end state s' as the starting state for the next fuzzing process, with changed fuzzing bias and/or operators.

6 Conclusion

With the proliferation of neural networks, in particular as action policies in dynamic environments, we believe that the AI planning community should be interested not only in how to learn such policies, but also in how to gain trust in them. Apart from explainability, certification is a key issue here. We believe that testing can be one key instrument for this purpose, in analogy to software testing. We have introduced a framework systematizing this approach, and discussed its basic implications and possibilities.

We view this work as a basis for discussing what could become a subarea in its own right. Interesting issues to explore include the implementation and exploration of our framework and simple fuzzing ideas in standard PDDL/PPDDL/RDDL settings and benchmarks; the in-depth exploration of bug-candidate generation and bug confirmation methods; the design of methods that treat a neural network policy as a white-box, biasing bug-candidate generation based on insights into the network's internal behavior. Finally, the link back to reinforcement learning should be made, leveraging identified bugs for targeted re-training.

7 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.

References

Alshiekh, M.; Bloem, R.; Ehlers, R.; Könighofer, B.; Niekum, S.; and Topcu, U. 2017. Safe Reinforcement Learning via Shielding. *CoRR* abs/1708.08611.

Bacchus, F. 2000. *Subset of PDDL for the AIPS2000 Planning Competition*. The AIPS-00 Planning Competition Comitee.

Bäckström, C. 1998. Computational Aspects of Reordering Plans. *Journal of Artificial Intelligence Research* 9: 99–137.

Carlini, N.; and Wagner, D. 2017. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy*, 39–57.

Clarke, L. A. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* 2(3): 215–222.

Coles, A. J.; Coles, A.; García Olaya, A.; Jiménez, S.; Linares López, C.; Sanner, S.; and Yoon, S. 2012. A Survey of the Seventh International Planning Competition. *The AI Magazine* 33(1).

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

Do, M. B.; and Kambhampati, S. 2003. Improving the Temporal Flexibility of Position Constrained Metric Temporal Plans. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS'03)*, 42–51.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-Black Planning: A New Systematic Approach to Partial Delete Relaxation. *Artificial Intelligence* 221: 73–114.

Domshlak, C.; and Mirkis, V. 2015. Deterministic Oversubscription Planning as Heuristic Search: Abstractions and Reformulations. *Journal of Artificial Intelligence Research* 52: 97–169.

Edelkamp, S. 2001. Planning with Pattern Databases. In *Proceedings of the 6th European Conference on Planning (ECP'01)*, 13–24.

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

Fern, A.; Yoon, S. W.; and Givan, R. 2006. Approximate Policy Iteration with a Policy Language Bias: Solving Relational Markov Decision Processes. *Journal of Artificial Intelligence Research* 25: 75–118.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20: 61–124.

Fulton, N.; and Platzer, A. 2018. Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, 6485–6492. AAAI Press.

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

Gehr, T.; Mirman, M.; Drachsler-Cohen, D.; Tsankov, P.; Chaudhuri, S.; and Vechev, M. T. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proceedings of the IEEE Symposium on Security and Privacy 2018*, 3–18. IEEE Computer Society.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence* 173(5-6): 619–668.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through Stochastic Local Search and Temporal Action Graphs. *Journal of Artificial Intelligence Research* 20: 239–290.

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.

Haslum, P.; and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS'00)*, 140–149. AAAI Press.

Hein, M.; and Andriushchenko, M. 2017. Formal Guarantees on the Robustness of a Classifier against Adversarial Manipulation. In *Proceedings of the 30th Annual Conference on Neural Information Processing Systems (NIPS'17)*, 2263–2273.

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

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

Hoffmann, J.; and Brafman, R. 2005. Contingent Planning via Heuristic Forward Search with Implicit Belief States. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS'05)*, 71–80. AAAI Press.

Hoffmann, J.; and Edelkamp, S. 2005. The Deterministic Part of IPC-4: An Overview. *Journal of Artificial Intelligence Research* 24: 519–579.

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.

Katz, G.; Barrett, C. W.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proceedings of the 29th International Conference Computer Aided Verification (CAV'17)*, volume 10426 of *Lecture Notes in Computer Science*, 97–117. Springer.

King, J. C. 1976. Symbolic Execution and Program Testing. *Communications of the ACM* 19(7): 385–394.

Klößner, T.; Torralba, Á.; Steinmetz, M.; and Hoffmann, J. 2021. Pattern Databases for Goal-Probability Maximization in Probabilistic Planning. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS'21)*. AAAI Press.

Könighofer, B.; Alshiekh, M.; Bloem, R.; Humphrey, L.; Könighofer, R.; Topcu, U.; and Wang, C. 2017. Shield synthesis. *Formal Methods in System Design* 51(2): 332–361.

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI'12)*, 540–545.

Lipovetzky, N.; and Geffner, H. 2017. A Polynomial Planning Algorithm that Beats LAMA and FF. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS'17)*, 195–199.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee.

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*.

Muise, C. J.; Belle, V.; and McIlraith, S. A. 2014. Computing Contingent Plans via Fully Observable Non-Deterministic Planning. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, 2322–2329. AAAI Press.

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.

Peot, M.; and Smith, D. E. 1992. Conditional Non-linear Planning. In *Proceedings of the 1st International Conference on Artificial Intelligence Planning Systems (AIPS'92)*, 189–197.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research* 39: 127–177.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *Artificial Intelligence* 193: 45–86.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description. Available at http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *Journal of Artificial Intelligence Research* 67: 129–167.

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.

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.

Trevizan, F. W.; Thiébaux, S.; and Haslum, P. 2017. Occupation Measure Heuristics for Probabilistic Planning. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS'17)*, 306–315.

Wong, E.; and Kolter, J. Z. 2018. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, 5283–5292. PMLR.

Younes, H. L. S.; Littman, M. L.; Weissman, D.; and Asmuth, J. 2005. The First Probabilistic Track of the International Planning Competition. *Journal of Artificial Intelligence Research* 24: 851–887.