Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2025)

On Picking Good Policies: Leveraging Action-Policy Testing in Policy Training

Jan Eisenhut[1], **Daniel Fišer**[2], **Isabel Valera**[1,3], **Jörg Hoffmann**[1,4]
[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2]Aalborg University, Denmark
[3]Max Planck Institute for Software Systems, Saarbrücken, Germany
[4]German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
{eisenhut,ivalera,hoffmann}@cs.uni-saarland.de, danfis@danfis.cz

Abstract

Testing is a natural approach to assess the quality of learned action policies π. Prior work introduced policy testing in AI planning as searching for *bugs* in π, that is, states where π is sub-optimal with respect to a given testing objective. Beyond quality assurance, an obvious application of these methods is policy selection: given several π to choose from, we can use testing to select the "least buggy" one. Here, we integrate testing-based policy selection into the training process. This includes making more informed decisions when selecting the final policy after training, as well as choosing more promising intermediate policies during the training process. Our experiments with ASNets action policies show that integrating testing allows us to more reliably obtain good-quality policies.

Source code and benchmarks —
github.com/fai-saarland/bughive/tree/icaps25-training

Introduction

Learned action policies π, particularly ones based on neural networks, are increasingly prevalent in AI planning (e.g., Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020; Ståhlberg, Bonet, and Geffner 2022; Wang and Thiébaux 2024; Rossetti et al. 2024). Prior work (Steinmetz et al. 2022; Eisenhut et al. 2023, 2024) introduced policy testing in classical planning as a means for quality assurance; we will refer to this as π-testing to avoid confusion with the test set in policy learning. A "bug" in a policy π is defined as a state s on which π is sub-optimal. Random walks are used to find π-test states, and sufficient criteria to prove sub-optimality (called π-test oracles) are used to prove π-test states to be bugs.

Here, we investigate the use of π-testing to enhance the policy learning process itself. We leverage π-testing for policy selection, an inherent sub-task of policy learning. Specifically, we investigate how to apply π-testing *offline*, as a validation criterion after training, as well as *online*, to focus on more promising candidates during training.

Regarding the offline application, say we have trained on smaller tasks for numerous epochs, possibly with several different initial weights or hyper-parameters. Saving all intermediate policies, we have accumulated many candidates. Which one to select is non-trivial as candidates flawless on the training set may be bad on the (unknown) test set, e.g. due to overfitting. A simple approach used in several prior works is to evaluate the candidates on a set of validation tasks. One can check, e.g., whether the candidates solve the initial states. Here, we propose π-testing as a means to enhance reliability in policy-candidate validation, minimizing the risk of choosing poor candidates. We realize this by considering additional states in each validation task, and assigning scores based on whether the policy solves them and whether we can classify them as bugs.

In our online use of π-testing, we affect not only the final policy selection but the candidate set itself. We introduce a training algorithm which, to the best of our knowledge, is the first one that does not always continue training on the most recent policy, and instead iteratively expands a tree-shaped policy-version history. All intermediate policies are stored in a set Θ, and in each epoch any policy can be selected for further training (so that, in particular, the same policy can have multiple successors). To inform the selection step, we employ π-testing, selecting the policy π from Θ based on its π-test result on the validation tasks. For comparison, we also run random selection from Θ in our experiments.

We experiment with ASNets policies (Toyer et al. 2020), on the IPC 2023 learning track benchmarks. Offline π-testing yields final policies with substantially better test set coverage. An additional observation here is that even the simple use of validation tasks – selecting the policy based on validation-set coverage, which the original ASNets authors do not report about – often yields a comparable performance boost. The main advantage of π-testing, compared to this, is that it yields more robust performance on states other than the IPC initial states, and in terms of the frequency of policy bugs (as found by the same π-testing machinery).

The results with respect to the online application of π-testing are mixed. π-testing based policy selection during training can yield improved performance when the budget in terms of training epochs is small; when that budget is high, performance is rarely improved and sometimes deteriorates. However, even though our results do not demonstrate a clear advantage across the board, the branching ability of our tree-shape training algorithm can be beneficial in some domains also for high budgets.

Background

We first provide the necessary background on classical planning, action policies represented by Action Schema Networks (ASNets) and their training, as well as policy testing.

Finite-Domain Representation Planning

A finite-domain representation **planning task** (Bäckström and Nebel 1995) Π is a tuple $\langle \mathcal{V}, \mathcal{A}, I, G \rangle$, where $\mathcal{V}$ is a finite set of **variables**, each $v \in \mathcal{V}$ associated with a finite domain $\mathrm{dom}(v)$; $\mathcal{A}$ is a finite set of **actions**; I is a full variable assignment called the **initial state**; and G is a partial variable assignment called the **goal**. Complete variable assignment over $\mathcal{V}$ are referred to as **states**. $\mathcal{S}$ denotes the set of all states and $s \in \mathcal{S}$ is a **goal state** if $G \subseteq s$. Each action $a \in \mathcal{A}$ is a triple $\langle \mathrm{pre}(a), \mathrm{eff}(a), \mathrm{cost}(a) \rangle$, where **precondition** $\mathrm{pre}(a)$ and **effect** $\mathrm{eff}(a)$ are partial variable assignments and $\mathrm{cost}(a) \in \mathbb{R}_0^+$ is the **action cost**. We say a is **applicable** in state s if $\mathrm{pre}(a) \subseteq s$. Applying an applicable action a to a state s results in a state $s[\![a]\!]$ such that $s[\![a]\!] = s$ except that all variables v on which $\mathrm{eff}(a)$ is defined are reassigned to $\mathrm{eff}(a)[v]$. Building on that, a sequence of actions $\vec{a} = \langle a_1, \ldots, a_n \rangle$ is applicable in $s_0 \in \mathcal{S}$ if there exist states $s_1, \ldots, s_n$ such that a_i is applicable in s_{i-1} and $s_i = s_{i-1}[\![a_i]\!]$. Applying $\vec{a}$ in s_0 results in $s_0[\![\vec{a}]\!] = s_n$. If $\vec{a}$ is applicable in s and $s[\![\vec{a}]\!]$ is a goal state, $\vec{a}$ is called a **plan** for s with cost $\mathrm{cost}(\vec{a}) = \sum_{i=1}^{n} \mathrm{cost}(a_i)$. A plan $\vec{a}$ is **optimal** if there exists no cheaper plan. The **cost-to-goal** function $h^\star$ maps each state s to the cost of an optimal plan for s if there exists a plan for s and to ∞ otherwise.

Action Policies and ASNets

Deterministic **action policies** $\pi \colon \mathcal{S} \to \mathcal{A} \cup \{\emptyset\}$ map states s to actions a applicable in s, or $\emptyset$ if no such action exists. We run π on s by iteratively applying the action selected by π until there either exists no applicable action, we reach a goal state, or we reach a state for the second time. Formally, the unique **run** $\sigma^\pi(s)$ of π on $s = s_0$ is the longest state action sequence $\langle s_0, a_0, \ldots, a_{n-1}, s_n \rangle$ consisting of pairwise different states and such that, for all $i < n$, $G \not\subseteq s_i$ and $a_i = \pi(s_i) \in \mathcal{A}$ is applicable in s_i with $s_{i+1} = s_i[\![a_i]\!]$. The **run cost** of π on s is $\mathrm{cost}^\pi(s) = \mathrm{cost}(\langle a_0, \ldots, a_{n-1} \rangle)$ if $G \subseteq s_n$ and $\mathrm{cost}^\pi(s) = \infty$ otherwise.

We focus on learned action policies represented by neural networks. While our proposed methods are generic, we implement them for the well-established ASNets architecture (Toyer et al. 2018, 2020). Leveraging the relation structure of planning problems, ASNets are domain-generalized policies, i.e., they are trained on small instances to learn a common set of weights θ, which are then used to instantiate ASNets policies for larger instances from the same domain.

ASNets are trained using so-called imitation learning: Before every training epoch, the current policy is used to sample states from the (small) training instances and for each such state s, an external (symbolic) planner is called to obtain the best action a for s. These pairs (s, a) of the input state s and its label a are then used as the training data, which (possibly) grow with every training epoch as more states are sampled and labeled by the external planner.

Algorithm 1: Original ASNets training algorithm.

Inputs: Training tasks Π_{train}
Params: Maximal number of epochs N_{epoch}
Output: Weights θ (policy model)

$\theta \leftarrow$ randomWeights();
$\mathcal{M} \leftarrow \emptyset$; // training data
for $i = 1, \ldots, N_{\mathrm{epoch}}$ **do**
  $\theta, \mathcal{M} \leftarrow$ trainEpoch$(\theta, \mathcal{M}, \Pi_{\mathrm{train}})$;
  if earlyTerminationCriterion$(\theta, \mathcal{M}, i)$ **then break**;

Algorithm 1 sketches the training algorithm of Toyer et al. The training data $\mathcal{M}$ is shared across all invocations of trainEpoch and extended in each epoch as described above. The trainEpoch procedure runs a fixed number of training cycles. In each cycle, it randomly samples a fixed-size mini-batch from $\mathcal{M}$ and uses the Adam optimizer to optimize weights θ. Training is stopped if the limit N_{epoch} is reached (we do not use a time limit as Toyer et al.), or if the trained policy solves all training instances in 20 consecutive epochs (which we denote here with the procedure earlyTerminationCriterion). The selected policy is always the result of the last trainEpoch invocation, so all intermediate policies can be discarded. In particular, this means that policy selection relies only on the training data $\mathcal{M}$, i.e., no validation data is used.

Policy Testing

We use the π-testing framework of Steinmetz et al. (2022) as adapted by Eisenhut et al. (2023, 2024). Given a planning task and an action policy π, a **bug** in π is a state s such that $\mathrm{cost}^\pi(s) > h^\star(s)$, i.e., π does not solve s even though s is solvable or π solves s but there exists a cheaper plan. The search for bugs in action policies is organized in a two-step procedure: we first build a pool $\mathcal{P} \subseteq \mathcal{S}$ of π-test states in a fuzzing process, and then we invoke π-test oracles on the generated pool states, attempting to identify them as bugs.

The **fuzzer** is based on random walks that can be guided by biases (Eisenhut et al. 2024), aiming to steer them towards bugs. While these can enhance π-testing, we do not employ biases here as they are policy-dependent and thus hinder a fair comparison of different action policies, i.e., we compare policies on the exact same pools.

Likewise, the **oracle** we employ is fixed in each comparison. We rely on bound-maintenance oracles (BMOs), the best-performing class of oracles of Eisenhut et al. (2023), which work by maintaining upper bounds on $h^\star$ across π-test states and decreasing them via comparisons based on dominance functions (Torralba 2017). They have the key ability to prove that π is sub-optimal on a π-test state without a planning process, allowing quick configurations, but can also be combined with more expensive (search-based) tools, yielding slower but more thorough oracles.

Applying Policy Testing in Policy Training

We present our refined training algorithm in Algorithm 2. It consists of three steps: a start-up phase based on the origi-

Algorithm 2: Advanced training including π-testing.

Inputs: Training tasks Π_{train}, validation tasks Π_{val}
Params: # random initializations N_{init},
start-up epochs N_{start} (per initialization),
main phase epochs N_{main},
maximal # expansions N_{exp} (per model)
Output: Weights θ (policy model)

$\mathcal{M} \leftarrow \emptyset$; // training data
$\Theta \leftarrow \emptyset$; // set of policy models θ
$\mathcal{T} \leftarrow \emptyset$; // map of models to π-test results
$\mathcal{E} \leftarrow \emptyset$; // map of models to # expansions
// start-up phase
for $i = 1, \ldots, N_{\text{init}}$ **do**
  $\theta \leftarrow$ randomWeights();
  $\Theta \leftarrow \Theta \cup \{\theta\}$; $\mathcal{T}[\theta] \leftarrow \pi$-test$(\theta, \Pi_{\text{val}})$; $\mathcal{E}[\theta] \leftarrow 0$;
  for $j = 1, \ldots, N_{\text{start}}$ **do**
    $\mathcal{E}[\theta] \leftarrow 1$;
    $\theta, \mathcal{M} \leftarrow$ trainEpoch$(\theta, \mathcal{M}, \Pi_{\text{train}})$;
    $\Theta \leftarrow \Theta \cup \{\theta\}$; $\mathcal{T}[\theta] \leftarrow \pi$-test$(\theta, \Pi_{\text{val}})$; $\mathcal{E}[\theta] \leftarrow 0$;
// main phase with online policy selection
for $i = 1, \ldots, N_{\text{main}}$ **do**
  $\theta \leftarrow$ selectOnline$(\Theta, \mathcal{T}, \mathcal{E}, N_{\text{exp}})$;
  $\mathcal{E}[\theta] \leftarrow \mathcal{E}[\theta] + 1$;
  $\theta, \mathcal{M} \leftarrow$ trainEpoch$(\theta, \mathcal{M}, \Pi_{\text{train}})$;
  $\Theta \leftarrow \Theta \cup \{\theta\}$; $\mathcal{T}[\theta] \leftarrow \pi$-test$(\theta, \Pi_{\text{val}})$; $\mathcal{E}[\theta] \leftarrow 0$;
// offline selection of final policy
$\theta \leftarrow$ selectOffline$(\Theta, \mathcal{T})$;

nal training algorithm, an optional main phase with *online* policy selection, and the *offline* selection of the final policy.

As inputs, we require a set of training tasks Π_{train} and validation tasks Π_{val}. Since ASNets training requires a high number of invocations of a teacher planner, we are restricted to training instances that a traditional planner can solve within seconds. In contrast, we are able to support significantly larger validation tasks. Their size is only limited by how quickly we can π-test policies on them meaningfully.

As in the original algorithm, $\mathcal{M}$ is the accumulated training data, including the set of states sampled from in each training epoch. Throughout the entire training process, we store all models θ in a set Θ. Each θ completely determines an action policy π_θ. We obtain a new θ either by sampling random weights (function randomWeights) or by loading an old model θ_{old} and training it for an epoch (function trainEpoch), in which case we call θ a successor of θ_{old} and the invocation of trainEpoch an expansion. Note that each expansion of the same θ_{old} likely yields a different successor, e.g., because $\mathcal{M}$ has changed or because random sampling from $\mathcal{M}$ results in different mini-batches.

Upon learning a new θ, we π-test the corresponding policy π_θ on Π_{val} and store the result in the map $\mathcal{T}$. In addition, we keep track of how often we have expanded each θ in a map $\mathcal{E}$, allowing us to balance the trade-off between focusing on the most promising models and exploring a higher number of them.

The start-up phase follows Algorithm 1, extending it only in that we conduct N_{init} random initializations of θ. We include it to guarantee a number N_{start} of training steps per random initialization. Apart from altering random seeds in each run, we could also adapt hyper-parameters such as the number of neural network layers. However, we decided against including that to keep our experiments simpler.

The main phase enables online π-testing based policy selection and branching. For N_{main} iterations, we select a model $\theta \in \Theta$ according to its π-test result $\mathcal{T}[\theta]$ and the number of expansions $\mathcal{E}[\theta]$. Then, we train θ producing a new set of weights, which we π-test on Π_{val} and add to Θ. The function selectOnline returns the best θ under the restriction that $\mathcal{E}[\theta]$ is below the expansion limit N_{exp}, i.e., θ with N_{exp} successors are ignored. Note that this online policy selection informed by π-testing on Π_{val} marks the only way in which we incorporate information derived from validation data in the training process. We never directly include any validation data in the training process itself.

Finally, we invoke selectOffline to select the final policy. The only algorithmic extension required for this offline policy selection is to store and π-test the intermediate policies. In particular, the number of expansions is ignored.

Policy selection in selectOffline and selectOnline is based on the same score, which consists of three equally weighted components derived from (a) the number of validation tasks where the policy solves the initial state, (b) the percentage of processed pool states solved by the policy, and (c) the percentage of them classified as bugs by a fixed oracle.[1] All three scores are relative, comparing policies to the best policy in each category. For example, if π_1 solves no validation task at all, π_2 solves only one, but one is the maximum, then π_1 receives the worst possible and π_2 the best possible score. However, if there is a π_3 solving all validation tasks, both π_1 and π_2 receive (almost equally) bad scores. In addition, to avoid full π-testing on larger tasks outside the scalability range of a policy π, we completely omit π-testing on a task if it fails to solve the initial state, in which case we base the scores on the assumption that π solves no states and all are bugs in π, even though we do not know whether this is true.

We have experimented with different test scores in preliminary experiments and found that the current set-up using equally weighted scores is a reasonable choice. In most cases, the percentage of solved pool states and identified bugs primarily act as tie-breakers in case multiple policies achieve (almost) the same number of solved validation tasks. We have seen that this leads to better results than focusing more on solved pool states or bugs (or even selecting the final policy based only on bug states).

Experiments

To conduct our experimental evaluation, we extend the framework of Eisenhut et al. (2024). Apart from using their π-testing tool, we implement our refined training algorithm within the ASNets framework introduced there, which is a faster re-implementation[2] of ASNets in C. To run π-testing

[1] Here, "processed" refers to the states that could be π-tested within a time limit of two minutes per task.

[2] https://gitlab.com/danfis/cpddl

Evaluating different selection criteria; all policies trained using "line" configuration (240 epochs)

Domain	$\lvert\Pi_{\text{test}}\rvert$	coverage (# solved instances) on Π_{test}					$\lvert\Pi_{\text{rel}}\rvert$	$\lvert\mathcal{P}\rvert$	% solved states $\in \mathcal{P}$			% bug states $\in \mathcal{P}$		
		early Π_{train}	loss Π_{train}	solved Π_{val}	π-test Π_{val}	solved Π_{test}			solved Π_{val}	π-test Π_{val}	solved Π_{test}	solved Π_{val}	π-test Π_{val}	solved Π_{test}
Blocksworld	60	39.3	42.3	**50.3**	**50.3**	52.5	48.5	4850	96.6	**96.8**	95.4	**3.0**	3.1	3.7
Childsnack	60	14.5	11.3	**35.8**	**35.8**	43.3	28.8	2706	**26.9**	**26.9**	**26.9**	**4.4**	**4.4**	4.6
Ferry	63	38.5	55.3	**63.0**	**63.0**	63.0	61.0	6100	**100.0**	**100.0**	**100.0**	1.0	**0.8**	1.0
Floortile	60	17.5	14.0	21.5	**23.0**	25.8	19.8	1975	20.1	20.1	**20.4**	3.0	3.1	**2.6**
Miconic	70	48.8	56.3	66.0	**70.0**	70.0	65.0	6062	99.3	**100.0**	96.9	66.7	**47.5**	69.3
Rovers	39	8.5	10.8	**12.8**	**12.8**	15.0	11.8	1175	**81.9**	**81.9**	81.3	39.8	39.8	**39.4**
Satellite	60	53.3	49.5	56.3	**58.8**	60.0	55.0	5500	95.5	**97.4**	96.4	39.4	34.4	**30.6**
Sokoban	43	4.5	4.8	**5.8**	**5.8**	7.0	5.0	458	83.4	**83.8**	78.1	34.6	**33.9**	37.6
Spanner	84	6.0	6.0	**81.8**	73.0	82.3	70.3	6122	**89.5**	85.3	88.7	49.5	**43.8**	49.2
Transport	55	37.0	34.0	38.8	**41.0**	42.3	37.8	3775	94.1	**98.2**	96.1	30.3	**21.4**	26.5

Evaluating policies trained using different algorithm configurations; same criterion for selecting final policy ("π-test Π_{val}")

Domain	$\lvert\Pi_{\text{test}}\rvert$	coverage (# solved instances) on Π_{test}														
			budget: 40 epochs				budget: 60 epochs				budget: 80 epochs				budget: 240 epochs			
		start	line	line π-test	tree	tree π-test	line	line π-test	tree	tree π-test	line	line π-test	tree	tree π-test	line	line π-test	tree	tree π-test
Blocksworld	60	20.3	44.3	**47.8**	35.8	43.8	47.0	48.5	43.0	**51.5**	44.8	49.5	49.3	**51.3**	50.3	52.0	**53.5**	53.0
Childsnack	60	16.0	29.5	**36.8**	32.5	32.8	18.8	36.8	**38.8**	35.3	**39.8**	38.3	39.3	36.5	35.8	**47.0**	42.0	39.3
Ferry	63	62.8	**63.0**	**63.0**	**63.0**	**63.0**	**63.0**	**63.0**	62.8	**63.0**	62.8	**63.0**	62.8	**63.0**	**63.0**	**63.0**	62.8	**63.0**
Floortile	60	0.0	16.3	**20.5**	3.0	3.5	**21.8**	19.0	6.8	4.3	**22.3**	20.3	12.0	8.8	**23.0**	18.8	22.0	19.8
Miconic	70	61.8	68.5	**69.3**	62.3	67.8	69.5	68.8	65.5	**70.0**	**70.0**	68.8	62.8	**70.0**	**70.0**	68.8	68.3	**70.0**
Rovers	39	0.0	**8.8**	8.5	3.8	1.8	**10.3**	**10.3**	4.5	9.0	**11.5**	10.3	6.3	9.3	**12.8**	10.3	11.0	10.8
Satellite	60	19.5	**58.0**	56.5	52.8	56.0	**58.5**	56.5	54.8	57.5	**57.8**	56.5	57.0	57.5	58.8	56.5	**59.3**	57.5
Sokoban	43	1.8	**5.5**	4.5	3.0	3.5	**5.8**	4.5	3.8	3.8	**5.3**	4.5	4.8	4.0	**5.8**	4.3	4.8	4.0
Spanner	84	66.8	74.3	67.0	72.8	**75.8**	**83.5**	67.0	79.0	75.8	**83.5**	67.0	78.3	73.8	73.0	67.0	**78.8**	69.0
Transport	55	25.8	**41.5**	38.5	28.5	39.3	**41.8**	39.0	37.3	38.0	**41.8**	38.8	37.5	38.0	**41.0**	38.8	40.8	40.3

Table 1: Policy quality in Π_{test}, across policies selected by different criteria (top) or trained differently (bottom). In the upper right part, $\Pi_{\text{rel}} \subseteq \Pi_{\text{test}}$ is the set of relevant problems considered for the comparison (all compared policies solve the initial state, π-testing completed). $\mathcal{P}$ is the joint state pool across all $\Pi \in \Pi_{\text{rel}}$. "% solved states $\in \mathcal{P}$" denotes the percentage of pool states solved by the respective policy and "% bug states $\in \mathcal{P}$" denotes the percentage of identified bugs among them. All results are averaged over four experiment runs.

during training (and for the overall experiment), we use an adapted version of lab (Seipp et al. 2017). The code and dataset is publicly available.

We take the benchmarks of the IPC 2023 learning track (IPC'23). For each domain, this contains 100 or slightly less (numbered) training tasks, out of which the first 20 form our training set Π_{train} and 40 are used as our validation set Π_{val} (we cover the full range of the original training set, i.e., if it contains exactly 100 tasks, every second of the remaining 80 tasks is selected for Π_{val}). Note that the IPC'23 training set already contains instances that are too large for imitation learning using a teacher planner, which limits the number of tasks we can select for Π_{train}. We adopt the IPC'23 test set Π_{test}, which consists of 90 tasks per domain, out of which 30 are classified as "easy","medium", and "hard", respectively. We omit tasks where pre-processing steps (computing the grounded task as used by ASNets or computing dominance functions for our π-test oracles) run out of memory. The final number of test tasks used for each domain is listed in Table 1, column $\lvert\Pi_{\text{test}}\rvert$.

During training, we store all policies on disk, which required only up to a few MiBs per policy. All policies are π-tested on precomputed state pools (of size up to 50/100 for validation/test tasks). For π-testing on Π_{val} (i.e., when selecting a policy), we use a quick BMO π-test oracle (Eisenhut et al. 2023) and a time limit of two minutes per task. When π-testing the final policies on Π_{test}, we use an extended BMO including the plan improver Aras (Nakhost and Müller 2010), and set time and memory limits of 12 hours and 8 GiB per task. The entire experiment (including training) is repeated four times with different random seeds and was run on a cluster of AMD EPYC 7702 processors.

To evaluate the *offline* application of π-testing, we compare five criteria for selecting the final policy, choosing:

1. π obtained after 20 consecutive epochs of solving all training tasks or the last π if no such policy exists ("early Π_{train}", resembling the selection criterion of Toyer et al. (2020) explained in the "Background" section);
2. π with the minimal loss on the training set as computed in trainEpoch ("loss Π_{train}");
3. π solving the most validation tasks ("solved Π_{val}");

4. π with the best π-test score ("π-test Π_{val}"); and
5. as a hypothetical best-case with access to Π_{test}, π solving the highest number of test tasks ("solved Π_{test}", timeout of 5 minutes per task)—this criterion is only added as a theoretical reference, which allows us to better assess the other methods.

Note that selecting the policy based on a loss on the validation set is not practicable, as the loss depends on the plan found by the teacher planner and Π_{val} contains tasks that are too large for the teacher planner to solve them in a reasonable time.

To evaluate the *online* use of π-testing, we compare different configurations of Algorithm 2 for multiple budgets in terms of the overall number of training epochs. We always use $N_{\text{init}} = 4$ seeds. "line" is the configuration closest to the original algorithm, using the entire budget in the start-up phase ($N_{\text{start}} \in \{10, 15, 20, 60\}$, $N_{\text{main}} = 0$). All other configurations use a short start-up phase ("start", $N_{\text{start}} = 5$) and the remaining budget in the main phase $N_{\text{main}} \in \{20, 40, 60, 220\}$). We include three such configurations:

1. "line π-test" which uses π-testing for policy selection but does not allow branching ($N_{\text{exp}} = 1$);
2. "tree" which allows branching ($N_{\text{exp}} = 4$) but uses random policy selection; and
3. "tree π-test" which uses branching and π-testing.

Table 1 shows the results. In the upper part, we evaluate the offline use of π-testing, comparing the said five criteria for selecting the final policy from a fixed set of 240 candidates trained with "line". Regarding coverage (left), the methods relying only on Π_{train} are significantly worse than the methods based on Π_{val}, which are basically on par with each other and often come close to "solved Π_{test}" (the theoretical optimum here, up to the different runtime restrictions). While "π-test Π_{val}" only has slight advantages over "solved Π_{val}" in some domains with respect to coverage, it has clear advantages when considering policy quality in more detail (right): In terms of solved states and bugs, the selection based on the π-test score is the best performing method, often even performing better than "solved Π_{test}" despite that method's access to Π_{test}. Also note that the table does contain information about plan length performance: in all but one domain, most pool states are solved and hence most identified bugs are states on which the policy's plan is sub-optimal. In general, the results show that employing a reliable policy selection method matters and that π-testing can serve to that effect.

In the lower part of the table, we evaluate the online application of π-testing, comparing the said four training configurations while fixing "π-test Π_{val}" for final policy selection. For small training budgets (particularly 40 epochs), π-testing based policy selection can yield improved coverage while for high budgets this is almost never the case. This trend is to be expected to a certain degree. It makes more sense to apply a bias to focus on particular candidates if we cannot consider a high number of them. Applying such a bias inevitably risks ignoring candidates and overfitting to Π_{val}. An aspect of our training algorithm that can potentially be useful for high budgets though is its ability to branch: for budget 240, "tree" is competitive with "line" in all domains (at least on par, or worse by a < 2 margin only), and is better by margins > 5 in Childsnack and Spanner (although in Childsnack, the ability to branch is clearly not necessary to reach a significant improvement, as "line π-test" outperforms both "tree" and "tree π-test" by ≥ 5). Given the unstable nature of ASNets training, we conjecture that random branching can constitute a form of regularization, yielding potential advantages of "tree" over "line" for high budgets.

A limitation of our experimental evaluation is that it does not include an analysis of the opportunity cost of running π-testing on Π_{val} versus extending the training set Π_{train} with additional tasks. It is clear that it can be practicable to extend Π_{train} to a certain degree and one could also generate additional training problems of comparable size. However, note we selected a setup in which Π_{val} contains instances too large for imitation learning using a teacher planner, limiting the potential of this alternative.

Conclusion

Learned action policies are gaining ever more traction. Here, we contribute a first integration of policy testing into the training process, leveraging π-testing scores for policy selection. Our results indicate that this can yield improved policies. An interesting topic for future work is to integrate π-testing into the training process more tightly, using found bug states to bias or guide the training priorities.

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

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS$^+$ Planning. *Computational Intelligence*, 11(4): 625–655.

Eisenhut, J.; Schuler, X.; Fišer, D.; Höller, D.; Christakis, M.; and Hoffmann, J. 2024. New Fuzzing Biases for Action Policy Testing. In *Proceedings of the 34th International Conference on Automated Planning and Scheduling (ICAPS'24)*, 162–167. AAAI Press.

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)*, 109–117. 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.

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.

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.

Rossetti, N.; Tummolo, M.; Gerevini, A. E.; Putelli, L.; Serina, I.; Chiari, M.; and Olivato, M. 2024. Learning General Policies for Planning through GPT Models. In *Proceedings of the 34th International Conference on Automated Planning and Scheduling (ICAPS'24)*, 500–508. AAAI Press.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS'22)*, 629–637. AAAI Press.

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)*, 353–361. AAAI Press.

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

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.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, 6294–6301. AAAI Press.

Wang, R. X.; and Thiébaux, S. 2024. Learning Generalised Policies for Numeric Planning. In *Proceedings of the 34th International Conference on Automated Planning and Scheduling (ICAPS'24)*, 633–642. AAAI Press.