Learner-Aware Instance Generation for Classical Planning via Neural Networks

Dániel Zoltán Bán, Dominik Ábel Sári

Department of Computer Science, Aalborg University, Denmark

Abstract

Automatic instance generation aims to automate planning problem creation, reducing reliance on hand-crafted generators. In this work we first evaluate NeSIG paired with CPDDL's ASNets and find that policies trained solely on NeSIG instances overfit - achieving high accuracy in-distribution but struggling on external benchmarks. To address this, we introduce two policy-aware difficulty metrics, Optimality Gap and Multi-Policy Consensus, which replace NeSIG's original planner-centric metric. By incorporating these metrics into a feedback loop between policy learning and instance generation, the modified NeSIG produces harder problems that expose policy weaknesses. Although initial experiments on the Miconic domain yielded limited improvements on external tests (likely due to computational constraints) trained policies showed greater robustness on increasingly challenging in-distribution sets. Our contributions include quantifying NeSIG's limitations and proposing a closed-loop, difficulty-guided framework to generate more informative training problems, laying the groundwork for enhanced policy generalization.

1 Introduction

Planning is a core area in machine intelligence, typically explored through relatively simple domains that can generalize to real-world applications. The problem instances in these domains serve as foundational "puzzles" for developing and evaluating planner models and algorithms. A robust collection of such instances is therefore very useful, as it does not only serve as a measure of the performance of certain solvers, but in many cases it is used as the training set of a machine learning model.

While many hand-crafted instances exist, improving these datasets requires significant manual effort. This makes automatic instance generation an increasingly important area of research. Earlier methods often involved randomly picking an initial state, then reaching the goal state through a random walk [3]. Some approaches combined this random walk with some semantic rules that ensured consistency [1]. These methods did make automatic generation easier; however, their randomness did not provide any control over the difficulty and diversity of the instances. In the approach of Marom & Rosman et al. (2020) [8] the instances are generated by applying backward search from a pre-defined goal condition. In this approach, the authors were able to control the difficulty of the generated problems by applying a heuristic for it in their search, but this method was limited to domains that have a structure with a single fixed goal. Using Machine Learning for instance generation had not been widely used method until the introduction of NeSIG [10], which marked a significant advancement in the field. NeSIG leverages Neural Logic Machines (NLMs) [2] that are neural networks designed for reasoning about logical inferences, making them well suited for tasks that involve relational structures, such as planning tasks. NeSIG utilizes NLMs, a solver to give feedback about the difficulty of the instances.

While the creators of NeSIG claim that their model is capable of creating diverse and difficult problems, there is a lack of research on the usefulness of NeSIG-generated problem instances, when it comes to training. Our motivation was to train a planner using NeSIG-generated problems, and through its evaluation find the weaknesses of the training set. With this information, it becomes possible to create an additional feedback loop in the NeSIG training process, that can further improve the diversity of its generated instances.

2 Background

2.1 PDDL

A PDDL (Planning Domain Definition Language) [6] problem consists of a planning domain description and the description of the specific instance. The domain describes the types of objects and predicates we can expect, and the actions we can use to achieve our goal. Each action has preconditions that must be true for the action to be applicable and effects which will be true after the action is executed. The instance specifies a set of objects, an initial state, and a goal condition that has to be reached. The solution of a planning problem is a sequence of actions that we can execute from the initial state, to reach a state where the goal condition is true.

For example, the Miconic domain models an elevator system: predicates describe relationships such as `(at ?p - passenger ?f - floor)` or `(lift_at ?f - floor)`, and actions include `(up ?f1 - floor ?f2 - floor)` (moving the elevator up from `?f1` to `?f2`), or `(board ?p - passenger ?f - floor)` (boarding passenger `?p` at floor `?f`), each with preconditions and effects. A Miconic instance then names specific floors and passengers, sets an initial configuration (elevator location and passenger locations), and defines a goal (e.g., each passenger delivered to a target floor).

2.2 NeSIG

For automated planning problem generation, we employ the Neuro-Symbolic Instance Generator (NeSIG) [10], a domain-independent framework that produces valid, diverse, and challenging problems. Unlike traditional, hand-crafted generators, NeSIG requires only a PDDL domain description, a set of user-defined consistency constraints, and a few generation parameters (e.g., maximum

problem size, number of problems to generate, etc.).

2.2.1 Generation process

NeSIG operates in two sequential phases, each governed by a learned policy. A Neural Logic Machine (NLM) policy starts from an empty or user-specified state and incrementally adds ground atoms. At each step, NeSIG enforces consistency constraints (e.g., an object cannot occupy two locations simultaneously) to ensure a valid initial state.

A second NLM policy applies domain actions to the generated initial state, exploring the reachable state space. From the final reachable state, a subset of atoms is selected to form the goal condition, ensuring solvability by design.

2.2.2 Instance evaluation

NeSIG evaluates instances based on three qualities:

- **Validity**
  A generated problem is valid if it is consistent and solvable. Consistency means that the initial state respects all domain constraints. These constraints are not described in the PDDL problem definition, but they are derived from common sense (such as an object cannot be at two places at once, or a logistics problem has to include at least one truck) and have to be formally specified before generating the instances. Solvability means that there is a solution for the problem. By design, NeSIG guarantees the validity of the problem.

- **Diversity**
  Diversity measures how different are the generated problems from each other. NeSIG measures diversity by mapping each problem to a feature vector of summary statistics (such as the number of objects and predicates of each type) and calculating the average and standard deviation of these vectors.

- **Difficulty**
  Difficulty refers to how hard a problem is to solve for a planner, measured by the number of nodes expanded by the planner. In NeSIG the planner used is Fast-Downward.

According to Núñez-Molina et al. (2023) [10] NeSIG is the leading framework for automated, adaptable problem generation in planning because it works across any domain, ensures each generated problem is valid, and focuses on creating problems that are both diverse and challenging.

2.3 Action Schema Networks

Action Schema Network (ASNet) is a neural network architecture introduced by Toyer et al. (2017) [13] specifically designed to learn generalized policies for both probabilistic and classical planning problems. ASNets show strong generalization capabilities across problem instances. This is achieved by mimicking symbolic reasoning for the PDDL domain it is training for, sharing weights based on action schemas and predicates, and enabling instance-invariant policies through its architecture.

ASNets are structured by alternating action layers and proposition layers. Action modules represent grounded actions, while proposition modules represent the grounded predicates. The connection between these modules, create a structure that reflects the domain's causal relationships. ASNets share weights among all modules using the same predicate symbol, helping to generalize independently from problem size. To further enhance the network's invariance to problem sizes, max-pooling is also applied over the inputs. To avoid vanishing gradients, skip connections are used, passing information through alternating layers more effectively. This helps in stabilizing the training; as in planning domains, effects often cascade, but this method can propagate these dependencies between modules well.

Each module gets a feature vector built from domain-level information. For action modules, this means an evaluation of whether the action is applicable for the current state and how often this action was selected in the past. In case of proposition modules, it is checked whether the predicate is currently true, whether it is part of the goal, and whether it appears in LM-cut landmarks. This usage of feature vectors allows ASNets to combine symbolic and learned features, so they are not learning from scratch, but are guided by symbolic planning heuristics, learning through mimicking symbolic reasoning.

These features of ASNets allow them to yield general policies even from small training sets if the training instances are diverse enough. However, ASNet performance may degrade if training instances lack critical schema combinations or object interactions. Therefore, ASNets serve as an appropriate learner for evaluating whether the instances generated by Nesig are diverse enough to allow general training of a policy.

2.4 CPDDL

CPDDL is a policy learning framework developed by Daniel Fišer [4]. It is not a learning algorithm by itself; however, it provides a framework that parses PDDL definitions, grounds domains, and converts problems into graph representations. These graph representations serve as the input of the neural architecture used by CPDDL, which is Action Schema Networks. Using the generalized learning capabilities of ASNets, CPDDL facilitates both policy training and evaluation.

Within our research, CPDDL functions as the evaluative component of the feedback loop integrated with NeSIG. Specifically, we use it assessing the diversity and training utility of generated instances by training policies and evaluating their performance on both NeSIG-generated datasets and external benchmark problems.

3 Problem Definition

To train a policy for an arbitrary planning domain, it is necessary to generate representative problem instances on which the policy can learn. Current instance-generation techniques fall into two broad categories:

1. Hand-crafted, domain-specific generators
   Researchers traditionally write ad-hoc, domain specific instance generators for each domain. Whenever a new domain is needed, or a domain is modified, its generator must be manually revised or rewritten, which is an error-prone, time-consuming process.

2. Domain independent instance generators
   Methods such as NeSIG eliminate the need for manual domain specific generators. Given a domain definition and a set of user-defined constraints it can already generate instances for that domain, thus significantly cutting back on the time and effort required to accommodate new or evolving domains.

While NeSIG represents a significant step toward domain-independent automatic instance generation, its efficiency as a training resource for learning generalized policies remains untested. In particular, it is unknown whether NeSIG-generated instances:

- Cover the full spectrum of possible instances needed to avoid blind spots in learners,
- Have sufficient diversity to promote generalization rather than overfitting.

In our paper we therefore have two primary objectives:

1. Instance-Generation Quality Assessment
   We will evaluate how well NeSIG, operating only on PDDL domain descriptions and user-specified constraints, can produce training sets that enable a learner (e.g., CPDDL) to generalize beyond the generated distribution. Performance will be measured both in-distribution (on other NeSIG instances) and out-of-distribution (e.g., standard PDDL-generators instances).
2. Enhancing the Instance Generator
   We aim to introduce modifications to NeSIG's generation process to produce instances that are optimally suited for policy learning. The goal is to generate a domain independent framework that generates problem instances and trains a problem-solving policy based on only a problem definition and a set of user-defined constraints

4 Methods

4.1 Iterative instance generation

While prior work established NeSIG's ability to generate problems exceeding ad-hoc generators in difficulty [10], its efficiency for training planning policies remains unverified. Therefore, we started out by assessing how can learners use NeSIG generated instances in their training, and how they perform on external test sets. Our evaluation specifically addresses whether NeSIG-generated instances enable robust policy learning and generalization to externally-created problems.

In order to eliminate the blind spots of policies trained on NeSIG-generated problems, we integrate CPDDL into NeSIG's generation loop and bias problem synthesis toward instances on which the current policy performs poorly. Precisely, our procedure alternates between (i) policy improvement via CPDDL and (ii) hardness-guided instance generation via NeSIG. Over successive iterations, the training set becomes enriched with progressively more challenging examples, yielding policies of increasing robustness. At each iteration CPDDL is trained on instances it previously failed to solve, forcing it to learn ways to resolve those weaknesses.

Algorithm 1 Iterative Instance Generation with CPDDL and NeSIG

1: Generate initial problem set p_1 using NeSIG with default settings
2: **for** each iteration **do**
3: **Policy training:** Train a policy on p_1 using CPDDL
4: **Integrate policy:** Update NeSIG's reward function to use the policy and a difficulty metric
5: **New instance generation:** Generate problem set p_2 using NeSIG with the updated reward
6: **Dataset augmentation:** Merge p_1 and p_2 to update the training set
7: **end for**

4.2 Difficulty metrics

In order to guide NeSIG towards generating instances that target the weaknesses of our policy, we define a quantitative difficulty metric to maximize, and replace NeSIG's original metric which was based on the number of nodes expanded by Fast-Downward. Our difficulty metric should assign higher scores to problems that are harder for the current policy to solve, thus encouraging NeSIG to target the policy's blind spots.

Throughout our experiments we implemented and evaluated two such metrics: Optimality Gap, and Multi-Policy Consensus. Both are designed to reflect how difficult an instance is for our policy, but they differ in how this is measured. Optimality Gap compares the policy's plan length against the plan length of Fast-Downward, while Multi-Policy Consensus aggregates results accross several saved policies to obtain a more general difficulty metric that reflects difficulty across multiple CPDDL-trained policies instead of a single one.

These metrics are used in the reward function of NeSIG, replacing the original difficulty function, while the components related to diversity and validity remain unchanged.

4.2.1 Optimality Gap

$$D = \begin{cases} M, & \text{if CPDDL fails} \\ \epsilon, & \text{if } l_{\text{FD}} \geq l_{\text{CPDDL}} \\ \dfrac{l_{\text{CPDDL}} + \epsilon}{l_{\text{FD}} + \epsilon} \cdot 100, & \text{if } l_{\text{FD}} < l_{\text{CPDDL}} \text{ and CPDDL succeeds} \end{cases}$$

Figure 1: Optimality gap difficulty metric

Where:

- D is the value of the difficulty
- M is a large constant representing a reward case
- ϵ is a small constant to avoid logarithm calculation and division with zero
- l_{CPDDL} is the plan length produced by CPDDL
- l_{FD} is the plan length produced by Fast Downward

The Optimality Gap difficulty metric (see Figure 2) is based on the idea of measuring the difficulty of the NeSIG generated instances by comparing the solution of a CPDDL policy with the solution of Fast-Downward on the specific problem. The difficulty metric is constructed in a way that distinguishes between three different cases. The most desired case, is when NeSIG generates a problem that the CPDDL policy cannot solve. In this case we assign a constant value C, that is a large number, in our experiments 500, which would realistically would never occur by simply solving a problem suboptimally compared to Fast-Downward. The second case is the least desired, when CPDDL does better, or equally as good as Fast-Downward - in this case the policy already performs well, so there is nothing to further learn from these instances. In this case a very low value is assigned, which is the constant epsilon value, that we use to avoid calculation with zeros. Finally in case CPDDL can solve the problem, but provides a longer plan than Fast-Downward, we still assign a smaller value, since even though CPDDL can solve the instance, there is still room for improvement. In this case we calculate the ratio of the length of the two plans, multiplied by 100, expressing the policy's solution length as a percentage of the optimal plan length. In this calculation, we also add epsilon to the plan lengths to avoid division by zero.

The biggest weakness of this difficulty metric was, that for NeSIG it takes multiple steps to get from empty instances, to non-trivial ones, and with this function there is no reward to be

	NeSIG trained		Pddl-generators trained	
	NeSIG test set	PDDL-Generators test set	NeSIG test set	PDDL-Generators test set
Blocksworld	100%	73%	100%	100%
Miconic	99%	13%	80%	13%
Visitall	99%	78%	-	-

Table 1: Average percentage of successfully solved test problems over the last 10 training epochs for each CPDDL model

gained by making the problem non-empty, but still trivial. In case of the old difficulty metric even for trivial instances, the reward would be higher for instances where there are some atoms, compared to completely empty instances. First we tried modifying the difficulty metric to separate the trivial problems and apply the old difficulty metric, to help the model getting to the generation of more complex problems. However, the separation of different difficulty calculations became misleading, as they were applied simultaneously. We therefore decided to proceed with pretraining the model using the old difficulty metric, for both of our rewards functions.

Originally instead of a constant reward for CPDDL failure, we used a dynamic reward based on the Fast-Downward plan length. However, when we started to rely on a pre-trained model, the length of the problems became less relevant when CPDDL already cannot solve them. Actually quite the opposite, the problems with smaller size that CPDDL policies cannot solve are the most valuable ones. Therefore, inspired during the monitoring of the multi-policy model, we decided to apply a constant high difficulty value in this case as well. Similarly with the majority of problems, where CPDDL performs good, the pre-training made it possible to set a hard threshold on every problem that is not hard for CPDDL, and assign a constant low difficulty value for them.

4.2.2 Multi-policy consensus

$$D = \sum_{i=1}^{n} \begin{cases} l_i + \epsilon, & \text{if CPDDL}_i \text{ succeeds} \\ M + \epsilon, & \text{if CPDDL}_i \text{ fails} \end{cases}$$

Figure 2: Multi-policy consensus difficulty metric

Where:

- D is the value of the difficulty
- l_i is the plan length generated by the policy i
- M is a large constant representing the rewards for failure
- ϵ is a small constant to avoid logarithm calculation with zero
- CPDDL$_i$ refers to the i-th policy
- n is the total number of policies

The Multi-policy difficulty metric (see Figure 3) relies solely on the solving capabilities of CPDDL policies. We start by picking a set of NeSIG trained CPDDL policies, from later parts of the training, when the performance has became stable. We choose the policies from across separate parts of the training, not only the very end, and we are trying to choose the highest performing policies. In our experiments we implemented this difficulty measure using i=5 policies.

For each generated problem, the CPDDL evaluation is called, using each policy. If a policy is able to solve the problem, the length of its plan is added to the difficulty. This way we propagate longer plans, even if CPDDL is able to solve them. In case a policy fails to solve a problem, a static difficulty score is added to the accumulated score. The addition of a negligible epsilon value also happens here, to avoid later calculations with zero.

Similarly to the Optimality Gap metric, here we also encountered issue that the training could not progress from generating trivial zero state instances so we also pre-trained the model as we did in the case of Optimality Gap. We observed that 50 steps of pre-training was enough to train the model to the level when it can generate complex problems. Then we started applying the multi-policy metric, which allowed the model to specialize for problems that are difficult for the CPDDL policies.

5 Experiments

5.1 Training CPDDL on NeSIG instances

5.1.1 Experimental setup

The first experiments we carried out were focused on examining how do NeSIG generated instances help a learner generalize, so it can also solve problems generated by different methods. We looked at three domains: Blocksworld, Miconic and Visitall. For all domains we trained CPDDL policies on NeSIG training set as well as on a PDDL-generators training set. Then all the policies were tested on both NeSIG and PDDL-generators test set. In case of Visitall, we were not able to obtain a usable large external test set, therefore we could not present results on how they perform as a training set. For the evaluation of the NeSIG trained Visitall model we used a set of unique problems from IPC Benchmark sets.

We assess the training utility of NeSIG-generated instances in two steps. First, problem instances are generated using NeSIG under controlled conditions: All experiments run for 24 hours on uniform hardware (1 GPU, 8 CPUs) with a fixed random seed (42) for reproducibility. Every 100 training steps, a set of 500 instances is saved, with the final set retained for analysis. Key parameters (max-init-actions and max-goal-actions) are calibrated to match IPC benchmark distributions.

Following instance generation, datasets were partitioned using an 80-20 training-test split via a bash script with randomized seeding. To mitigate selection bias, multiple training sets were created by varying split seeds across experiments.

We then trained policies using CPDDL's Action Schema Networks (ASNets) implementation (see section 2.4) under default configurations. Model snapshots were saved at each epoch during training on NeSIG-generated instances.

To evaluate the policies we used two test sets. One was the NeSIG-generated test set from the 80-20 split. The other one was a set of external ad-hoc generated problems, to test whether the training was enough to generalize over test sets generated by different methods. To test the policies, CPDDLs evaluation function was used, which logs the success rates for each policy.

In Table 1 you can see the average performance of the last 10 policies. The numbers indicate the percentage of problems that the policies were able to solve.

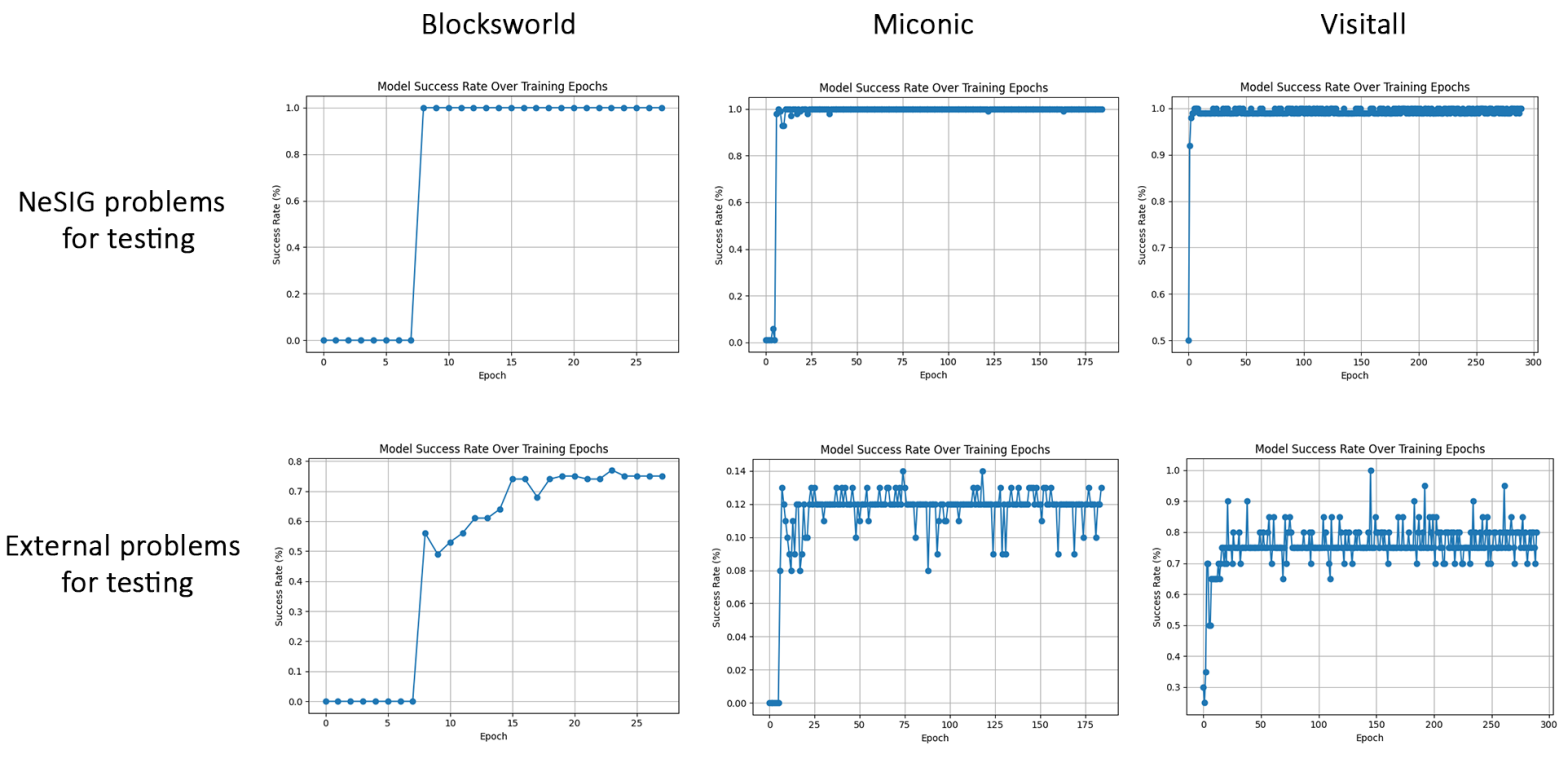

Figure 3: Policy performance over training epochs trained on NeSIG problem set for three domains

5.1.2 Results

The first thing that becomes clear from the results is that all the policies trained on NeSIG training set are performing almost perfectly on NeSIG test sets, solving almost 100% of the test problems already after a few epochs of training. Training on PDDL-generators instances does not result in such good performance during testing on instances generated by its own method.

However, when it comes to testing the NeSIG policies on PDDL-generators test set, the performance greatly drops. This gap in the performance is illustrated well on Figure 3. This suggests that the instances generated by NeSIG result in overfitting in the policies, and do not enable the solver to generalize over unseen problems that are generated by different methods. In certain cases, such as on the Miconic domain, the performance is significantly worse, the policy solving only roughly 13 percent of the test problems.

Another interesting result of the experiments was how the PDDL-generators trained policies performed when seeing NeSIG problems. While they performed similarly badly on the PDDL-generators test set, the NeSIG test set seemed to be a lot easier to solve for the policies, solving around 80% of the test instances for Miconic, and reaching even 100% for Blocksworld. This raises some questions regarding the claim in the NeSIG research paper, about the higher difficulty of problems NeSIG can achieve compared to PDDL-generators. It seems like while NeSIG optimizes its instances' difficulty on Fast-Downward's expanded nodes for each plan, the generated problems are not that difficult to solve for a learner like CPDDL-ASNets, trained on simple PDDL-generators problems. Based on the results of these experiments we can conclude, that NeSIG's instance generation method has room for improvement, when it comes to the usefulness of generated instances for training a solver. The diversity of the generated instances can be optimized more, so the trained policy avoids overfitting, and the generated problems cover more edge cases that the solver can learn.

5.2 Testing NeSIG instances after feedback

5.2.1 Experimental setup

To measure whether the modified NeSIG difficulty metric improved the quality of problems generated for training, the similar experiments were carried out on the newly trained CPDDL policies as for the original NeSIG trained models. Using the modified difficulty metrics described in the Methods section, we trained new NeSIG generation models to extend the original train and test sets. Then we measured the performance of the different NeSIG trained models on these problems sets. With the newly trained NeSIG models we saved 500 instances each, applied the 80-20 split on them and extended the original problem sets respectively. This resulted in the extended problem sets being half old instances and half new instances.

To see whether the performance of the trained solver improved using the new training data, we evaluated the newly trained models on the extended NeSIG generated test set, as well as the external test sets used before. In Table 2 you can see the average performance of the last 10 training epochs of the CPDDL policies. Similarly to the previous experiments, the numbers indicate the percentage of problems that the policy was able to solve. Due to limited time and resources, we can only present results on the Miconic domain.

5.2.2 Results

First of all, we tested how is the performance of the original NeSIG trained CPDDL model on this extended test set, and the performance of the model degraded, as before it was able to solve all problems that came from NeSIG. This is not suprising, as the modified version of NeSIG was designed to create problems that this specific model cannot solve. Already during the monitoring of the NeSIG training we could observe that the Optimality Gap difficulty metric resulted in more unsolvable problems, meaning that some of the newly added problems in case of the Multi Policy reward are still regular problems for the CPDDL model. Despite this, we believe that longer training for the NeSIG generator with

	Original NeSIG test set	NeSIG extended test set	PDDL-generators test set
Original NeSIG training	99%	92% (multi) 54% (opt)	13%
Multi Policy Consensus training	100%	98% (multi)	15%
Optimality Gap training	100%	78% (opt)	12%

Table 2: Performance of CPDDL policies after one iteration on the Miconic domain

the Multi Policy function would have eventually yielded similarly good results as the Optimality Gap.

As it can be seen on the results, there was no significant increase in performance of the policies on the external test sets. In case of the Multi Policy based reward, there is a slight increase, this might be related to the fact that this training set was containing less very hard problems, which prevented the model from getting stuck too much with hard problems.

Finally, while the original NeSIG trained policies achieve almost perfect scores on their own NeSIG test sets in our first experiments, both the Multi Policy and the Optimality Gap models are performing worse on the test set of their own generators'. Despite this, they both perform perfectly on the original simple NeSIG test sets, and outperform the original NeSIG model when it comes to these broader and harder updated test sets. On Figure 4 this improvement in problem solving capability can be seen, on the respective datasets of each modified difficulty metric. Overall this still shows some improvement in the capabilities of these models compared to the original, confirming that with this way of instance generation the NeSIG instance sets can be improved.

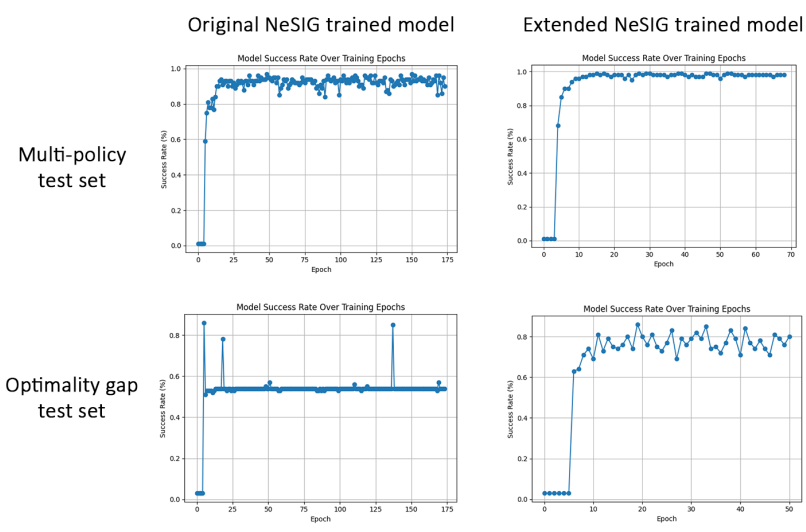

Figure 4: Performance of the policies trained on the original and extended NeSIG problem sets - The extended models were tested on their own test sets respectively

6 Related works

Beyond our approach, there are different takes on improving policies, that utilize the information about the weaknesses of a generated training set, and improve the instance generator model. In most cases, this feedback to the generator model is done manually by humans by identifying blind spots of the system. However, more efficient methods have emerged in the past years that can offer automatization, or shed light on issues in the policies that humans could not identify. These solutions are relevant to our work both as motivation to create a similar system, but they can also be good inspiration for future work.

6.1 Curriculum learning

Curriculum learning by Narvekar et al. (2020) [9] is a method where the training data fed into the learner progressively gets harder, aligned with the capabilities of the model. This ensures that the model we are training keeps seeing problems that are hard to solve, but it is trying to keep the problems within the capabilities of the model in a controlled fashion. Having such a well adjusted training set can help keeping the training stable and it can prevent overfitting by constantly increasing difficulty outside of the learners current comfort zone.

There are three main topics within curriculum learning that Narvekar mentions. The first and least researched is the instance generation, more precisely the control over the difficulty, diversity and solvability of generated problems. This is the most relevant for our research, as Nesig could potentially be used in a curriculum learning setup, and this method could also be beneficial in the implementation of an automated feedback loop between Nesig and the learner it is generating problems for, similarly to a GAN architecture. There are other related studies that have more focus on the generation of problems, Justesen et al. (2018) [7] explores procedural level generation, applying a feedback loop that allows a dynamic difficulty control based on the agent's performance. Florensa et al. (2017) [5] uses reverse curriculum generation, increasing the difficulty by setting the initial state further and further away from the goal step by step.

The other two main topics within curriculum learning are sequencing and knowledge transfer. Sequencing ensures that the problems are sorted by difficulty in a way that it results in the best training. Knowledge transfer is the problem of how learning from one task can help solving another. While these aspects could potentially be interesting to investigate when building such an automated system, they are less relevant for our work.

6.2 Policy debugging

Policy debugging is a systematic evaluation of learned policies, to identify blind spots and gaps in the capabilities of the policy, that lead to poor performance. Steinmet et. al (2022) [11] investigates this approach on action-policy testing, where the test states for the policy are generated through fuzzing search, then the bugs are confirmed through oracles. A bug in this context means a state where the the current policy performs suboptimally, either not finding the cheapest plan (quantitative bug) or not finding the goal while possible (qualitative bug). The generation of test cases to reveal the bugs is done by fuzzing, performing random walks from states that are included in the original plan of the solver. This walk can be biased towards low-quality policy decisions or unusual states, testing the performance of the policy starting from states it would not reach by itself.

Once the set of test states are created, oracles are used to check whether the state is a bug and reveal a weakness in the policy. There are several types of oracles, optimized for different kind of bugs, depending on what is the optimal plan we are aiming for. They can optimize for the lowest cost plan, look ahead to see whether better outcomes are reachable, or just simply considering not reaching the goal a bug. If the oracle determines that the

policy performs suboptimally, that reveals a weakness in the policy, and this information can be utilized for the further training of the policy, or at the creation of training sets for the training of a new policy.

This is where the method of policy debugging becomes relevant for our work with NeSIG, as this approach could potentially be used for revealing the blind spots of policies trained on NeSIG generated instances. This information can be valuable in making NeSIG better at generating instances that are useful for training.

6.3 Autoscale

Autoscale [12] is an automated framework for generating benchmark sets in classical planning. Given a planning domain and its instance generator, Autoscale selects sequences of parameter configurations to produce instances with smoothly scaled difficulty, ensuring the resulting benchmark covers easy, medium, and hard problems while minimizing bias toward specific planners. Unlike NeSIG - which generates individual instances - Autoscale focuses on curating sets of instances that collectively enable robust planner evaluation.

This approach complements our work with NeSIG and CPDDL. While our current iterative loop generates instances targeting policy weaknesses, Autoscale could also be a good fit at generating instances covering a full difficulty spectrum to ensure policies generalize broadly. Future work could integrate NeSIG as Autoscale's instance generator, using CPDDL-trained policies as planners. Such integration could further mitigate overfitting and improve generalization in policy learning.

7 Discussion

7.1 Discussion

The results indicate that integrating CPDDL feedback into NeSIG's instance-generation loop effectively produces training sets rich in problems exposing CPDDL's weaknesses. While the original NeSIG generator created instances that CPDDL solved easily, the modified approach shifted focus to instances where CPDDL either fails or returns suboptimal plans. The training sets generated by our method contained a higher concentration of difficult instances, demonstrating that we successfully crafted a "better" training set tailored to CPDDL's blind spots.

However, when evaluating the resulting CPDDL policies on the external test set, performance did not consistently surpass the baseline. One contributing factor may be that CPDDL training was halted before full convergence due to limited computational resources. Therefore, it is possible that given additional time or hardware, the model could have learned a better policy from the instances we generated. This is especially true for the Multi-Policy based training, where the final set did not contain as many hard instances as the Optimality Gap training set did. This however might also result in more optimal training, as we did not investigate the ideal concentration of hard instances in a training set.

Each of the difficulty metric had distinct benefits: pretraining on Fast-Downward expansions accelerated the generation of nontrivial instances, multi-policy consensus highlighted shared failures across CPDDL policies, while not being affected by optimality, and the optimality gap also emphasized instances on which the policy creates a suboptimal plan, however neither metric alone guaranteed broad improvements.

Despite the test results do not indicate significant increase in performance on external test set, the increase in difficulty and diversity among the generated instances is clear. The problem solving capabilities of the new generation of CPDDL policies increased compared to the original, and this could possibly further increase with several iterations. This, however, would also require surpassing certain limits of the learning capabilities of CPDDL, by increased computational resources or using different parameters. We came to this conclusion after examining the training logs of CPDDL, as it showed the learning difficulties of the model over the harder training set.

7.2 Future work

A natural next step is to validate the framework across additional domains (such as Logistics or Satellite) to determine whether similar patterns emerge. New domains with more complex interactions might also reveal types of blind spots not captured by our current metrics, motivating enhanced difficulty functions.

It is also essential to assess whether CPDDL itself is a limiting factor. Repeating the iterative instance-generation process with alternative learners would clarify whether the lack of performance gain stem from the data or is it is a limitation of CPDDL.

Another promising direction is a two-stage training process combining multi-policy consensus and optimality gap. Starting with multi-policy consensus could help in training a policy that solves nearly every instance (though not optimally). Afterwards switching to optimality gap would encourage the generation of problems that push the policy toward optimal plans. Exploring different approaches for these transitions could reveal how to balance discovering new edge cases with refining solution quality.

Finally, in our method we kept NeSIG's diversity reward untouched, but we might also have some opportunities for improvements by revisiting how the diversity reward gets calculated. By guiding NeSIG to generate a wide range of instances in addition to solver-hard instances, we could reduce overfitting while still addressing CPDDL's specific weaknesses.

In summary, although our method clearly produces more informative training sets by targeting CPDDL's weak spots, converting those sets into consistently stronger policies may require addressing both computational constraints and learner bottlenecks, exploring new domains, and refining difficulty metrics.

8 Conclusion

This work has investigated the efficiency of automatic, domain-independent instance generation for training generalized planning policies, focusing on the Neuro-Symbolic Instance Generator (NeSIG) and its interaction with CPDDL's Action Schema Networks (ASNets). Our first set of experiments demonstrated that, although NeSIG-generated problems are valid, diverse, and challenging when evaluated against Fast-Downward, they have limitiations when used as a training set for CPDDL policies, yielding near-perfect performance on in-distribution instances but poor generalization to externally generated benchmarks. By analyzing this shortcoming, we identified that NeSIG's original difficulty metric (based solely on Fast-Downward's node expansions) does not mean instances will also be difficult for learning algorithms, and make a good training set.

To address this limitation, we introduced two alternative difficulty metrics - Optimality Gap and Multi-Policy Consensus - that explicitly target instances where existing CPDDL policies either fail or produce suboptimal plans. Integrating these metrics into NeSIG's reward function resulted in progressively richer training sets containing instances that are harder for CPDDL to solve. As a consequence, the modified generator produced problem distributions that challenged CPDDL, and found it's weak points more effectively. Although our evaluation on the Miconic domain did not

show a significant improvement on external test sets after a single feedback iteration - likely constrained by computational resources or by CPDDL's ability to generalize well for a domain - it did reveal that the policies trained on difficulty-guided NeSIG data exhibit higher robustness on more challenging in-distribution sets. This suggests that continued iterations of the feedback loop, coupled with longer training schedules or alternative policy learners, could yield policies with superior generalization properties.

Overall this work has two main contributions. First, we have provided a quantitative assessment of NeSIG's baseline utility for policy learning, identifying clear gaps between planner-specific difficulty and learner-specific generalization. Second, we have demonstrated that augmenting NeSIG's reward structure with learner-specific difficulty metrics can steer instance generation toward problems that more directly address a policy's weaknesses. While further work is necessary to confirm these findings across additional domains and with different learners, our results suggest that using a feedback loop with better difficulty metrics can be a useful way to generate better training sets that help planning policies improve in more meaningful and general ways.

Acknowledgment

We would like to express our sincere gratitude to our supervisor, Alvaro Torralba, for his continuous support, valuable feedback, and for generously agreeing to supervise our work despite our delayed schedule. His insights and guidance were a great help throughout the project.

We also thank Daniel Fišer for his assistance with using and understanding CPDDL, which helped us resolve several issues we encountered during our experiments.

References

[1] Tomás de la Rosa and Raquel Fuentetaja. Bagging strategies for learning planning policies. *Ann. Math. Artif. Intell.*, 79(4):291–305, 2017.

[2] Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. *CoRR*, abs/1904.11694, 2019.

[3] Alan Fern, Sung Wook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pages 191–199. AAAI, 2004.

[4] Daniel Fišer. Cpddl. https://gitlab.com/danfis/cpddl, 2023.

[5] Carlos Florensa, David Held, Markus Wulfmeier, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. *CoRR*, abs/1707.05300, 2017.

[6] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019.

[7] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Procedural level generation improves generality of deep reinforcement learning. *CoRR*, abs/1806.10729, 2018.

[8] Ofir Marom and Benjamin Rosman. Utilising uncertainty for efficient learning of likely-admissible heuristics. In J. Christopher Beck, Olivier Buffet, Jörg Hoffmann, Erez Karpas, and Shirin Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, pages 560–568. AAAI Press, 2020.

[9] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *J. Mach. Learn. Res.*, 21:181:1–181:50, 2020.

[10] Carlos Núñez-Molina, Pablo Mesejo, and Juan Fernández-Olivares. Nesig: A neuro-symbolic method for learning to generate planning problems. In Ulle Endriss, Francisco S. Melo, Kerstin Bach, Alberto José Bugarín Diz, Jose Maria Alonso-Moral, Senén Barro, and Fredrik Heintz, editors, *ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*, volume 392 of *Frontiers in Artificial Intelligence and Applications*, pages 4084–4091. IOS Press, 2024.

[11] Marcel Steinmetz, Daniel Fiser, Hasan Ferit Eniser, Patrick Ferber, Timo P. Gros, Philippe Heim, Daniel Höller, Xandra Schuler, Valentin Wüstholz, Maria Christakis, and Jörg Hoffmann. Debugging a policy: Automatic action-policy testing in AI planning. In Akshat Kumar, Sylvie Thiébaux, Pradeep Varakantham, and William Yeoh, editors, *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24, 2022*, pages 353–361. AAAI Press, 2022.

[12] Álvaro Torralba, Jendrik Seipp, and Silvan Sievers. Automatic instance generation for classical planning. In Susanne Biundo, Minh Do, Robert Goldman, Michael Katz, Qiang Yang, and Hankz Hankui Zhuo, editors, *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, pages 376–384. AAAI Press, 2021.

[13] Sam Toyer, Felipe W. Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. *CoRR*, abs/1709.04271, 2017.