Per-Domain Generalizing Policies for Classical Planning: On Scaling Behavior and Validation Instances

Anonymous submission

Abstract

Recently, graph neural networks have emerged as a powerful tool in the AI planning community to train per-domain generalizing policies. The key objective is scaling the policy's capabilities from small training instances to large test instances. However, prior work has used fixed sets for both testing and validation. We first demonstrate that fixed test sets are insufficient for accurately assessing scaling behavior and, therefore, introduce refined methodology for evaluating scaling behavior. Second, we introduce a method generating validation sets dynamically, on the fly, increasing instance size so long as informative and feasible, leading to improved policy selection. Our method consistently improves scaling behavior of graph neural network policies across 9 domains.

1 Introduction

Recently, *graph neural networks* (GNNs) have emerged as a powerful tool within the Artificial Intelligence (AI) research community. GNNs excel at capturing complex relationships and dependencies in graph-structured data. Besides others, they have demonstrated remarkable effectiveness in predicting molecular structures (Gilmer et al. 2017), facilitating drug discovery (Bongini, Bianchini, and Scarselli 2021), and enhancing recommender systems (Fan et al. 2019).

AI planning is a well-established field of the AI community, which, at its core, is about devising sequences of actions, so-called *plans*, to achieve specific goals in problems. This process is typically applied across various *instances* of a planning *domain*, aiming to identify the optimal sequence of actions that transitions from the instances' well defined start state to meet its established goal conditions. Practical applications of planning include network penetration testing (Speicher et al. 2019), autonomous driving (Bai et al. 2015), construction planning (Li and Lu 2019), and enterprise risk management (Sohrabi et al. 2018).

Given the structure of planning problems and the rule-based manner in which the AI planning community typically formulates them, AI planning appears to be well-suited for the application of GNNs. Consequently, researchers have begun leveraging GNNs for learning policies that generalize across *all* instances of a planning domain, so-called *per-domain generalizing policies*. The key objective in this setting is *scaling behavior*, i.e., the ability to generalize from small training instances to large test instances.

Toyer et al. presented the pioneering approach, utilizing a CNN-inspired architecture to compute general policies for probabilistic planning (2018). Their architecture features modules specific to predicates and action schemas. Rivlin, Hazan, and Karpas utilized a novel architecture combining layers of both GNNs and so-called Transformers, addressing several classical planning problems through training with deep reinforcement learning (2020). Furthermore, Ståhlberg, Bonet, and Geffner developed a relational GNN architecture for learning general value functions for classical planning problems (2022a; 2022b).

A commonality of these works is that they evaluate their approaches on limited test sets. For example, Ståhlberg, Bonet, and Geffner evaluate their approaches on test sets from the *International Planning Competition* (IPC), using up to 120 test instances (Ståhlberg, Bonet, and Geffner 2022a,b, 2023, 2024). Rivlin, Hazan, and Karpas also use IPC domains using 50 test instances, and Toyer et al. consider three different probabilistic planning domains with up to 30 instances per domain (Toyer et al. 2018, 2020a). While such test sets may be suited for evaluating planning algorithms, we argue that they are not representative enough to asses the scaling behavior of per-domain generalizing policies.

As an illustrative example, consider Figure 1. We here investigate the performance of a policy trained using Ståhlberg, Bonet, and Geffner's architecture (2022a). The x-axis sorts the instances of the IPC 2023 Blocksworld domain according to their *size*, e.g., the number of objects. The y-axis measures the performance as the average percentage of solved instances, which we refer to as *coverage*. The number of instances in the IPC test sets is limited, typically only one or two instances per size and for some there are none at all. Thus, the policy's coverage for both IPC easy (orange) and medium (brown) mostly reaches 100% or 0%, meaning it can either solve all or none of the few instances of a certain size. In contrast, consider the blue curve which shows the results of our novel *scaling behavior evaluation* method. Instead of relying on limited test sets, we leveraged the publicly available IPC generators to generate test instances as needed. Specifically, we ran the policy on generated instances of increasing size up until the average coverage reached a confidence interval with a certain probability. Our evaluation method clearly shows how the policy's cov-

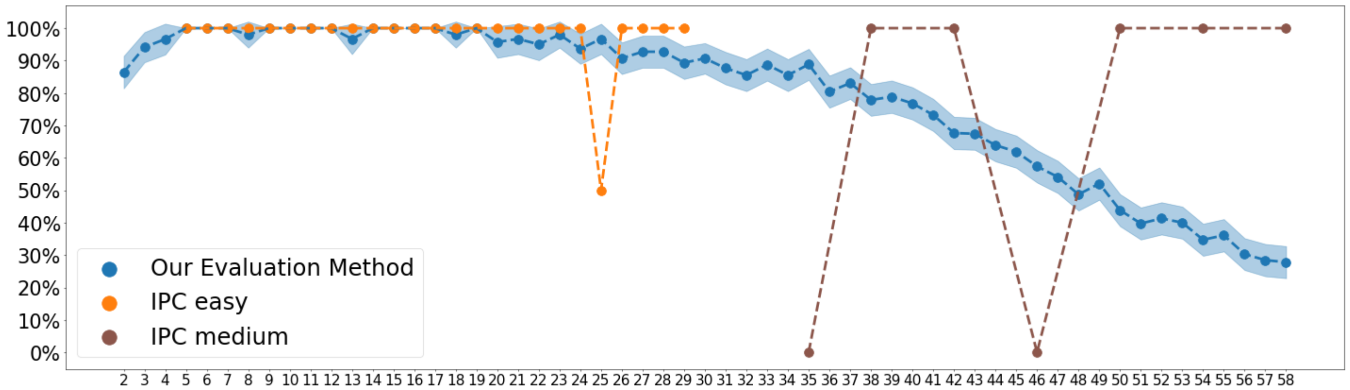

Figure 1: Scaling behavior (average coverage over instance size) of a Blocksworld policy trained following (Ståhlberg, Bonet, and Geffner 2022a), measured on IPC test sets compared to our evaluation method. The confidence intervals are shown as light blue shade.

erage decreases steadily with increasing instance size. However, this key insight is not visible in the IPC test sets at all.

Following this motivation, we give a detailed introduction of our scaling behavior evaluation in Section 3, which allows to systematically and accurately assess scaling behavior of policies. Further, we introduce two measures for summarizing how far and how well policies generalize.

The problems caused by a limited number of instances not only occur during testing but also during the validation phase of training. During validation, the current policy's performance is assessed by computing the loss (Ståhlberg, Bonet, and Geffner 2022a) or the coverage (Rossetti et al. 2024) on a small pre-computed validation set. When the training ends, the final policy is selected as the one that performed best during validation. However, as limited validation sets may estimate scaling behavior inaccurately, the existing validation approaches may fail to select the best policies found during training.

Therefore, instead of using pre-computed validation sets, in Section 4 we introduce *dynamic coverage validation*, which dynamically generates increasingly large validation instances to better estimate scaling behavior.

In Section 5, we run experiments with GNN policies trained following (Ståhlberg, Bonet, and Geffner 2022a) on 9 IPC domains. Using our scaling behavior evaluation method, we show that dynamic validation consistently improves performance across all but one of these domains, with substantial advantages in 7 of them.

2 Relational Graph Neural Network

Before introducing the newly proposed evaluation, we briefly explain the GNN architecture used throughout this paper. We use Ståhlberg, Bonet, and Geffner's relational graph neural network (R-GNN) (2022a; 2022b; 2024). However, we do not use the same policies, but use the R-GNN architecture to train new policies.

For a fixed domain, the R-GNN is trained to compute a *state value function* V, which, given a state s from any instance, returns the optimal cost $V(s)$ for reaching a goal state from s. The learned state value function V then induces a policy if, for any given state s, we always transition to the successor state s' with the lowest state value $V(s')$.

The R-GNN consists of several multi-layer perceptrons (MLP): A predicate MLP_p for each predicate p, an update MLP_U, and an output MLP_O. Algorithm 1 shows the steps of the R-GNN's forward pass. Given a state s, the R-GNN computes an embedding $h(o)$ for each object o in the state, which are then used to predict $V(s)$. The object embeddings $h(o)$ are computed iteratively, where the initial embeddings $h^0(o)$ only contain zeros. Every iteration i consists of two steps:

1. *Message Computation*: For each atom $q := p(o_1, ..., o_n)$ in s, we pass the involved objects' embeddings $h^{i-1}(o)$ to MLP_p, which computes a so-called message $m_{q,o}$ for each object $o \in q$.
2. *Embedding Update*: For each object o, we aggregate the messages $m_{q,o}$ into a single message m_o by applying the smooth maximum function across each dimension. The new embedding $h^i(o)$ is then computed by passing $h^{i-1}(o)$ and m_o to MLP_U, with the addition of a residual connection.

After L iterations, we aggregate the object embeddings $h^L(o)$ into a single state embedding $h(s)$ by computing a dimension-wise sum. Lastly, the state embedding $h(s)$ is given to MLP_O, which outputs $V(s)$.

3 Scaling Behavior Evaluation

This section introduces our refined methodology for evaluating scaling behavior. Additionally, we introduce two measures for summarizing the evaluation results.

Systematic instance size scaling. As discussed in Section 1, the commonly used IPC test sets are not well suited for the systematic evaluation of policy scaling behavior due to their limited number of test instances. Much work has been done in the past on benchmark instance scaling for the purpose of evaluating planning systems (e.g., (Hoffmann et al. 2006; Torralba, Seipp, and Sievers 2021)). Here, we merely require a systematic scheme to generate instances

Algorithm 1: Relational graph neural network.

Input: Set of atoms q in state s, objects $o \in s$
Output: State value $V(s)$

```
1 h^0(o) ← 0^k for each object o ∈ s ;
2 for i = 1, ..., L do
3   for each atom q := p(o_1, ..., o_n) do
4     m_{q,o_j} ← [MLP_p(h^{i-1}(o_1), ..., h^{i-1}(o_n))]_j ;
5   for each object o do
6     m_o ← aggregate({{m_{q,o} | o ∈ q}}) ;
7     h^i(o) ← h^{i-1}(o) + MLP_U(h^{i-1}(o), m_o) ;
8 h(s) ← Σ_{o∈s} h^L(o) ;
9 V(s) ← MLP_O(h(s)) ;
```

Algorithm 2: Scaling behavior evaluation.

Input: Policy π, instance generator $\mathcal{G}$, CSP
Parameters: Statistical parameters ϵ and κ, plan length bound L, coverage threshold τ, consecutive fails threshold ζ
Output: Statistical coverage $\hat{\mathcal{C}}_n$ per instance size n

```
1  n ← 0 ;
2  fails ← 0 ;
3  while fails < ζ do
4    n ← n + 1 ;
5    if not instanceOfSizeExists(n) then continue ;
6    Ĉ_n ← −∞ ;
7    i ← 0 ;
8    possibleInp = CSP(n, ∞) ;
9    while P(|Ĉ_n − C_n| > ε) < κ do
10     Inp ← uniform(possibleInp) ;
11     I ← generateInstance(G, Inp) ;
12     R_i ← runPolicy(π, I, L) ;
13     i ← i + 1 ;
14     Ĉ_n ← Σ_{j=1}^{i} R_j / i ;
15   if Ĉ_n < τ then
16     fails ← fails + 1 ;
17   else
18     fails ← 0 ;
```

of scaling size. In designing such a scheme, we stick to community conventions and existing instance generators as much as possible.

We define instance size as the number of objects. This leaves open the question of *which* objects, i.e., given a desired size n, how to compose the object universe from the different sub-types. Our answer is a uniform distribution over the possible compositions given the respective IPC instance generator. Obtaining the possible compositions is non-trivial as IPC instance generators often do not allow to directly set the number of objects of any given type (requiring, e.g., to instead set the x- and y-dimensions of a map), and often implement implicit assumptions across object types (e.g., at least one truck per city). We capture these constraints in terms of CSP encodings, and use a CSP solver to find valid generator inputs that yield instances of size n.

Specifically, we model instance size as a constraint $n = c_1 v_1 + \cdots + c_k v_k + c_0$, where v_i are the CSP variables encoding the generator's arguments, and the constants c_i capture the numbers of objects created by the generator. We represent any implicit assumptions made by the generator as additional constraints. These CSP encodings tend to be very small, and CSP solving time is negligible.

For example, in Childsnack the task is to prepare and serve different kinds of sandwiches to children. The generator parameters are v_1 number of children, v_2 trays, v_3 sandwiches. The generator always adds $c_0 = 3$ tables, as well as bread and content objects for each child yielding $c_1 = 3$; it returns an error if there are fewer sandwiches than children. Accordingly, our CSP encoding is $n = 3 \cdot v_1 + v_2 + v_3 + 3 \wedge v_1 \leq v_3$.

For the purpose of generating an individual instance in dynamic validation, we compute all solutions to the CSP, sample one of these uniformly, and pass it as input to the generator. [1] If some of the generator parameters do not affect instance size, for instance the ratio of allergic children in Childsnack, we sample their values uniformly from the possible range.

Our algorithm. Algorithm 2 outlines our evaluation method.

[1] In dynamic coverage validation (see Section 4), to limit computational cost, we only compute the first 100 solutions to the CSP and sample from these uniformly.

We start with instances of size $n = 1$, so as to evaluate policy performance across the entire domain. In the algorithm, we skip over values of n for which no domain instance exists according to the generator parameters and assumptions (and hence our CSP is unsolvable). We only stop after ζ consecutive failures to meet the coverage threshold τ, to allow for temporary lapses in policy performance.

For instance generation, we draw uniformly from all possible size-n instances. $CSP(n, \infty)$ returns all solutions to the CSP for size n and $\mathcal{R}_i$ is a Boolean whose value is 1 iff the policy found a plan. We impose a plan length bound L on policy executions, which we compute for each domain automatically as $L = 3N$ where N is the average length of the teacher plans on the largest training instances. Further, we add the current instance size n to L, as the optimal plan length typically increases with the number of objects. We keep generating instances and running the policy until the estimated performance for size n is within a confidence interval – precisely, with parameters κ and ϵ, until we reach a confidence of $(1 - \kappa)$ that the error between average coverage $\hat{\mathcal{C}}_n$ and real coverage $\mathcal{C}_n$ is at most ϵ. We refer to the resulting value $\hat{\mathcal{C}}_n$ as *statistical coverage*. The algorithm outputs that value as a function of n.

As an example, consider Figure 2, which depicts the evaluation results of two policies trained on the Blocksworld domain. The statistical coverages $\hat{\mathcal{C}}_n$ per instance size n are plotted as curves, and the shaded areas represent the 5%-confidence intervals.

Summary measures. When analyzing the scaling behavior of per-domain generalizing policies, mainly two characteristics are of interest: how far a policy scales, i.e., up to which size it can reliably solve instances, and how good it

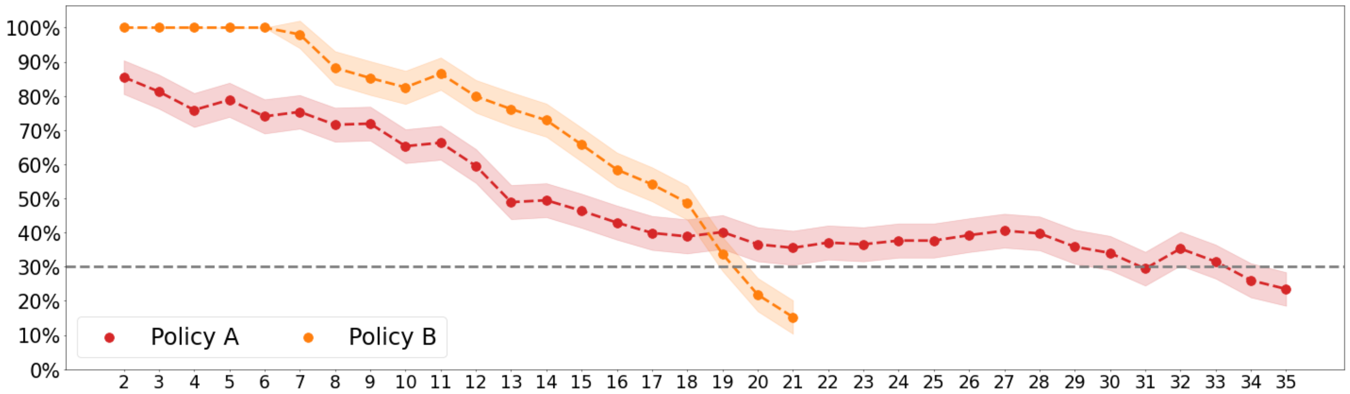

Figure 2: Statistical coverages $\hat{\mathcal{C}}_n$ of two policies trained on the Blocksworld domain using the R-GNN architecture.

generalizes, i.e., how many instances it solves. While both can be observed from visualizing the statistical coverages (e.g., see Figure 2), we here introduce two numerical measures to support this investigation.

The *Scale* measure is the largest instance size n before the policy falls below the threshold τ for ζ consecutive times, measuring "how far up" we get a minimum level of performance. The *SumCov* sums up the statistical coverages $\hat{\mathcal{C}}_n$ up to instance size n, measuring the "area beneath the coverage curve".

Consider Figure 2 again. Policy A (red) scales better, reaching the termination criterion (dashed line) only after size 33, while policy B (orange) has a Scale value of 19. However, policy B achieves higher statistical coverages before termination, yielding a SumCov value of 16.1, whereas policy A has a lower SumCov value of 15.4. This example shows that it is important to consider both measures.

4 Dynamic Validation

We next introduce our dynamic validation method. We give an overview of prior work, and then describe the method itself.

Prior work. Per-domain policy learning typically relies on a form of supervised learning (Ståhlberg, Bonet, and Geffner 2022a,b, 2024; Müller et al. 2024; Rossetti et al. 2024), training the policy to imitate an optimal planner on a set of small training instances. To identify when the policy achieves the best scaling behavior – generalization to larger domain instances – it is validated after every epoch, assessing its current performance on a set of larger validation instances. From all policies encountered during this process, the one with best validation set performance is selected as the final policy. Algorithm 3 outlines this training loop.

For the validation in line 6 of this loop, a common approach is to compute a loss between the policy's predictions and a teacher planner's decisions (Ståhlberg, Bonet, and Geffner 2022a,b, 2024). Alternatively, the policy can be validated by running it on the validation instances and computing coverage, i.e., the fraction of solved instances, which has the benefit of not requiring to run the teacher on the validation set (Rossetti et al. 2024).

Algorithm 3: Per-domain policy training loop.

Input: Training set T, validation set V, epochs E
Output: Policy π_{best}

```
1 π_0 ← random ;
2 π_best ← π_0 ;
3 v_best ← 0 ;
4 for i = 1, ..., E do
5     π_i = train(π_{i−1}, T) ;
6     v_i = validate(π_i, V) ;
7     if v_i better than v_best then
8         π_best ← π_i ;
9         v_best ← v_i ;
```

All prior approaches to validation in per-domain policy learning do, to the best of our knowledge, rely on a predefined fixed validation set. Yet this limits their ability to assess scaling behavior. The data on a fixed validation set is not informative if the policy already has perfect coverage/loss there. Further, the fixed validation sets are typically taken from IPC instance suits, limiting the number of available validation instances and hence the ability to see fine-grained differences between policies.

These limitations are quite unnecessary. As we discuss next, one can generate validation instances on the fly, ensuring informativity for policy selection as well as feasibility of the validation process.

Dynamic validation. Algorithm 4 outlines our dynamic validation method. The overall mechanics of the procedure are similar to scaling behavior evaluation, with the main difference being that it does not provide statistical guarantees to reduce computational cost.

Given training instances of maximal size n_0, we generate validation instances starting at $n_0 + 1$. We keep generating m instances of each size so long as policy coverage remains above a threshold τ, otherwise we terminate immediately. As in scaling behavior evaluation, we impose a plan length bound $L = 3N$, but we do not increase L based on the current instance size n. The final validation score v_π of policy π is computed as the sum of achieved coverages $\mathcal{C}_i$.

Algorithm 4: Dynamic coverage validation.

Input: Policy π, instance generator $\mathcal{G}$, CSP, size n_0
Parameters: per-size #instances m, plan length bound L, coverage threshold τ
Output: Validation score v_π

```
1  n ← n_0 ;
2  repeat
3      n ← n + 1 ;
4      if not instanceOfSizeExists(n) then
5          C_n ← 0 ;
6          continue ;
7      possibleInp = CSP(n, 100) ;
8      for i ∈ {1, ..., m} do
9          Inp ← uniform(possibleInp) ;
10         I ← generateInstance(G, Inp) ;
11         R_i ← runPolicy(π, I, L) ;
12     C_n ← Σ_i R_i / m ;
13 until C_n < τ;
14 v_π ← Σ_{i=n_0+1}^{n} C_i ;
```

5 Experiments

Our experiments evaluate our dynamic validation method for GNN policies, against loss-based and coverage-based validation on fixed validation sets as used in prior work. We employ our scaling behavior evaluation methods to obtain fine-grained comparisons. In what follows, we outline our benchmarks, training and validation set construction, policy training and selection setup, and empirical results.

Benchmarks. We use 9 different domains, which is a common number for papers on per-domain policy learning (e.g., Rivlin, Hazan, and Karpas 5 (2020), Ståhlberg, Bonet, and Geffner 8 (2022a), 9 (2022a), 10 (2023)). 7 of our domains have already been used in the context of per-domain generalization, while we additionally use Ferry and Childsnack from the latest IPC (2023). The generators were taken from IPC 2023 where available, and otherwise from the FF domain collection.[2]

Training. We use (Ståhlberg, Bonet, and Geffner 2022a)'s R-GNN architecture. The R-GNN learns a state value function and the policy is obtained by greedily following the best action, i.e., the action leading to the state with the lowest state value. To prevent cycles, we prohibit the policy from visiting states more than once (Ståhlberg, Bonet, and Geffner 2022b).

We use similar training hyperparameters as Ståhlberg, Bonet, and Geffner with 30 GNN layers, a hidden size of 32, a learning rate of 0.0002 for the Adam optimizer (Kingma 2014), and a gradient clip value of 0.1 (Ståhlberg, Bonet, and Geffner 2022a,b). However, we use a fixed number of 100 training epochs and a batch size of 1024 for fast training. For each domain, the training was repeated with three random seeds.

Instance sets. We construct the instance sets by uniformly sampling 100 instances per size from a range of 11 sizes, discarding duplicates. For each domain, except Visitall, we assign the instances of the eight smallest sizes to the training set. Similarly, for the fixed validation sets, we randomly select 12 instances, equally distributed (if possible) among the three largest instance sizes. The instance sizes used for each domain are presented in Table 1.

We note that strictly separating instance sizes of training and validation sets is critical for generalization. Without this separation, the policy with the best validation performance may be the one that has only learned to generalize up to the largest instance size shared between both the training and validation sets.

Domain	Training	Validation
Blocksworld	$[7-14]$	$[15-17]$
Ferry	$[8-15]$	$[16-18]$
Satellite	$[8-15]$	$[16-18]$
Transport	$[8-15]$	$[16-18]$
Childsnack	$[8-15]$	$[16-18]$
Rovers	$[10-17]$	$[18-20]$
Gripper	$[8-15]$	$[16-18]$
Visitall	$\{4, 9, 16, \dots, 49\}$	$\{64, 81\}$
Logistics	$[8-15]$	$[16-18]$

Table 1: Number of objects in instances used for training and validation sets.

We use the *seq-opt-merge-and-shrink* configuration of Fast Downward (Helmert 2006) with limits of 20 minutes and 64 GB as the teacher planner, computing optimal plans for both sets. We discard instances for which the planner fails to find a plan within the given time and memory limits. Additionally, we terminate the plan generation early if the planner fails to find a plan for 10 consecutive instances.

Joint policy training and selection. Our setup is designed such that we perform the training procedure jointly for all three validation methods. After every training epoch, we apply each method in turn, fixed-set loss (henceforth: loss), fixed-set coverage (henceforth: coverage), and our dynamic-set coverage method as introduced in Section 4 (henceforth: dynamic coverage). At the end of training, for each validation method, we select the respective best policy. In this manner, we guarantee that any differences in policy performance *are exclusively due to the difference in validation methods*.

[2] https://fai.cs.uni-saarland.de/hoffmann/ff-domains.html

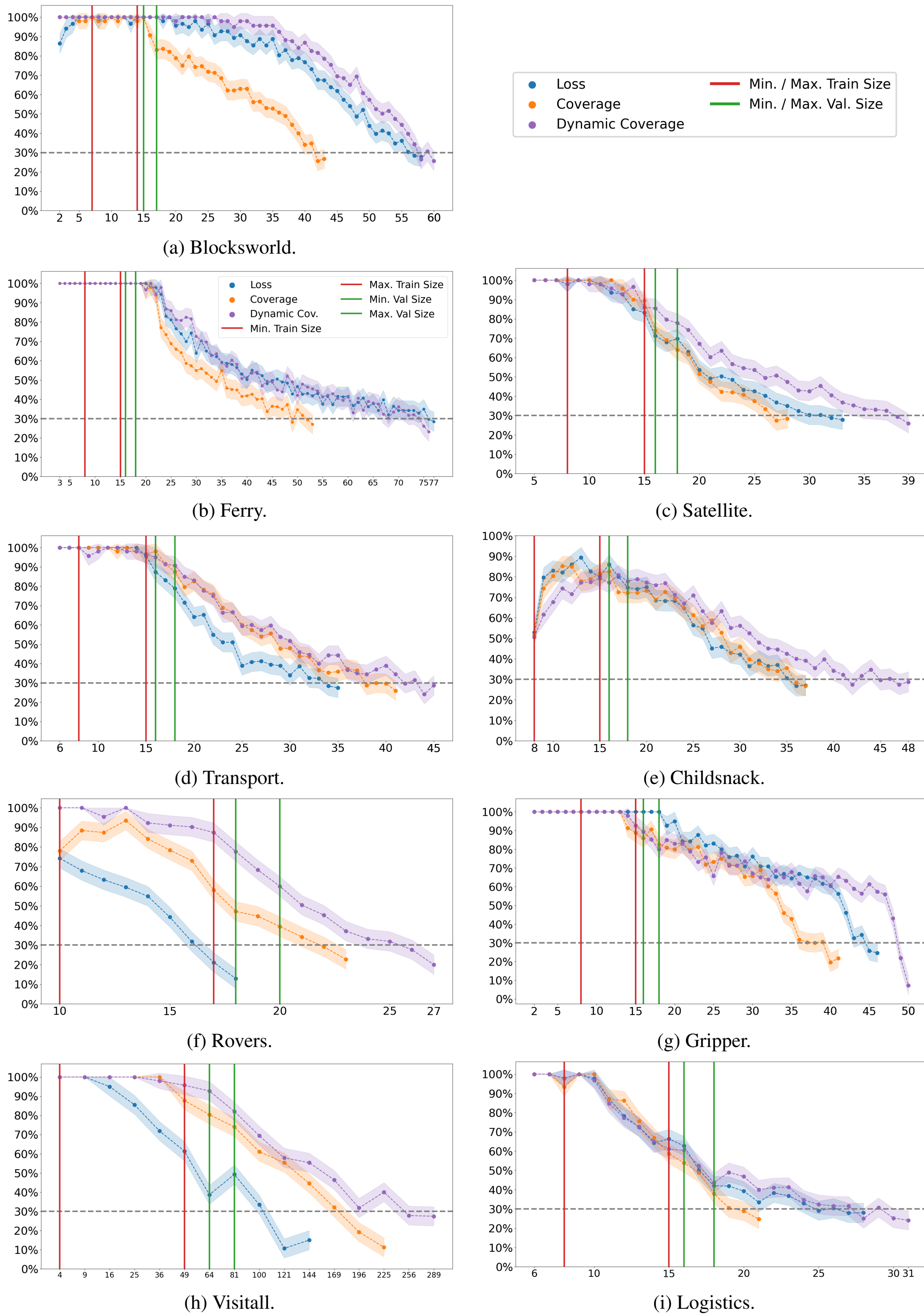

Figure 3: Statistical coverage $\hat{\mathcal{C}}_n$ over n of policies selected using fixed-set loss (blue), fixed-set coverage (orange), and dynamic coverage (purple) validation on 9 domains. The instance sizes used for training are within the vertical red lines, the instance sizes used in fixed validation sets are within the two green lines. Dynamic covarage validation starts at the lower green line (in Visitall that line has a gap to the largest training size because there are no instances for the sizes in between).

	Loss		Coverage		Dynamic	
Domain	Scale	SumCov	Scale	SumCov	Scale	SumCov
Blocksworld	56	44.47	41	30.43	**58**	**48.61**
Ferry	**75**	46.55	51	33.9	74	**46.63**
Satellite	31	18.15	26	16.37	**37**	**22.29**
Transport	33	19.4	39	23.72	**43**	**25.56**
Childsnack	35	17.62	35	17.63	**46**	**22.16**
Rovers	16	3.96	21	8.07	**25**	**11.6**
Gripper	44	35.15	39	29.35	**48**	**36.54**
Visitall	100	6.46	169	9.55	**225**	**10.97**
Logistics	26	13.02	19	10.39	**29**	**14.16**

Table 3: Scale and SumCov scores of policies selected using loss, coverage, and dynamic coverage validation.

For each instance size, dynamic coverage validation generates $m = 10$ instances and stops when the coverage drops below $\tau = 30\%$. In coverage, we imposed the same fixed plan length bound L as in dynamic coverage. Note that, during scaling behavior evaluation, L is scaled linearly with the instance size for all policies. The plan length bounds for each domain are listed in Table 2. Further, we imposed a one-hour time limit on the validation processes, but this was never reached in our experiments.

Domain	**Validation**	**Evaluation**
Blocksworld	123	$123 + n$
Ferry	72	$72 + n$
Satellite	42	$42 + n$
Transport	36	$36 + n$
Childsnack	18	$18 + n$
Rovers	54	$54 + n$
Gripper	69	$69 + n$
Visitall	162	$162 + n$
Logistics	30	$30 + n$

Table 2: Plan length bounds used during validation and evaluation.

Results. Figure 3 shows the scaling behavior evaluation of the policies validated based on loss (blue), coverage (orange), and dynamic coverage (purple). The confidence intervals were computed with $\epsilon = 0.05$ and $\kappa = 0.1$, and the evaluation stopped when the estimated coverage dropped below $\tau = 30\%$ for $\zeta = 2$ consecutive times.

The policies selected by the two fixed-set validation methods have roughly comparable performance, each outperforming the other in 2 domains and being close in the others. Dynamic validation, however, consistently yields the best scaling behavior across all domains, except Ferry where it is close to the best method; it exhibits substantial advantages in Satellite, Transport, Childsnack, Rovers, Gripper, Visitall, and Logistics.

To provide a summary view of these results, consider Table 3, which lists the Scale and SumCov measures for all policies. Reflecting Figure 3, our dynamic validation method dominates in all domains in both measures, with the single exception of Scale in Ferry.

Remember that in these results the policy selection was performed on the same training run, so the superiority of dynamic validation is exclusively due to policy selection.

6 Conclusion

Per-domain generalization in classical planning is a natural and popular setting for policy learning. Our work contributes new insights into the evaluation of scaling behavior for empirical performance analysis and the use of validation for policy selection in this context. The results are highly encouraging, showing clear improvements in 8 out of 9 domains.

An intriguing aspect of these improvements is that they are obtained through *policy selection* exclusively. We plan to investigate ways to feed back insights from validation into training, guiding the training process towards better scaling behavior. Another interesting direction is the application of our ideas in training processes based on reinforcement learning instead of supervised learning (e.g., (Rivlin, Hazan, and Karpas 2020; Ståhlberg, Bonet, and Geffner 2023)). Since our methods are agnostic to the policy representation, all this can in principle be done in arbitrary frameworks (e.g., (Toyer et al. 2020b; Rossetti et al. 2024)).

Another promising direction for future research is to extend our generalization evaluation to analyze additional policy properties, such as average plan length. We note that similar approaches have been successfully employed in the context of deep reinforcement learning (Gros et al. 2023, 2024).

Regarding further scientific experiments, an interesting analysis could be to vary the size of the training data as well, and determine its impact on scaling behavior. We could, for example, examine Scale and SumCov scores as a function of training data size.

Lastly, further constraints could be imposed on the instance generation process, as the IPC generators used in this work may generate instances that are large yet trivial, e.g., a Transport instance with 500 trucks but only one package. The resulting instances may be better suited for learning per-domain generalization, or enable more fine-grained evaluation.

References

Bai, H.; Cai, S.; Ye, N.; Hsu, D.; and Lee, W. S. 2015. Intention-aware online POMDP planning for autonomous driving in a crowd. In *2015 ieee international conference on robotics and automation (icra)*, 454–460. IEEE.

Bongini, P.; Bianchini, M.; and Scarselli, F. 2021. Molecular generative graph neural networks for drug discovery. *Neurocomputing*, 450: 242–252.

Fan, W.; Ma, Y.; Li, Q.; He, Y.; Zhao, E.; Tang, J.; and Yin, D. 2019. Graph neural networks for social recommendation. In *The world wide web conference*, 417–426.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*, 1263–1272. PMLR.

Gros, T. P.; Groß, J.; Höller, D.; Hoffmann, J.; Klauck, M.; Meerkamp, H.; Müller, N. J.; Schaller, L.; and Wolf, V. 2023. DSMC Evaluation Stages: Fostering Robust and Safe Behavior in Deep Reinforcement Learning–Extended Version. *ACM Transactions on Modeling and Computer Simulation*, 33(4): 1–28.

Gros, T. P.; Müller, N. J.; Höller, D.; and Wolf, V. 2024. Safe Reinforcement Learning Through Regret and State Restorations in Evaluation Stages. In *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*, 18–38. Springer.

Helmert, M. 2006. The Fast Downward Planning System. 26: 191–246.

Hoffmann, J.; Edelkamp, S.; Thíebaux, S.; Englert, R.; Liporace, F.; and Trüg, S. 2006. Engineering Benchmarks for Planning: the Domains Used in the Deterministic Part of IPC-4. 26: 453–541.

Kingma, D. P. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Li, D.; and Lu, M. 2019. Classical planning model-based approach to automating construction planning on earthwork projects. *Computer-Aided Civil and Infrastructure Engineering*, 34(4): 299–315.

Müller, N. J.; Sánchez, P.; Hoffmann, J.; Wolf, V.; and Gros, T. P. 2024. Comparing State-of-the-art Graph Neural Networks and Transformers for General Policy Learning. In *ICAPS Workshop on Planning and Reinforcement Learning (PRL)*.

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized planning with deep reinforcement learning. *arXiv preprint arXiv:2005.02305*.

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 International Conference on Automated Planning and Scheduling*, volume 34, 500–508.

Sohrabi, S.; Riabov, A.; Katz, M.; and Udrea, O. 2018. An AI planning solution to scenario generation for enterprise risk management. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Speicher, P.; Steinmetz, M.; Hoffmann, J.; Backes, M.; and Künnemann, R. 2019. Towards automated network mitigation analysis. In *Proceedings of the 34th ACM/SIGAPP symposium on applied computing*, 1971–1978.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022a. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 629–637.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022b. Learning generalized policies without supervision using gnns. *arXiv preprint arXiv:2205.06002*.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2023. Learning general policies with policy gradient methods. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 647–657.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2024. Learning General Policies for Classical Planning Domains: Getting Beyond C _2. *arXiv preprint arXiv:2403.11734*.

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS'21)*, 376–384.

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020a. Asnets: Deep learning for generalised planning. *Journal of Artificial Intelligence Research*, 68: 1–68.

Toyer, S.; Thiébaux, S.; Trevizan, F. W.; and Xie, L. 2020b. 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 AAAI Conference on Artificial Intelligence*.