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

Timo P. Gros[1,2], **Nicola J. Müller**[1,2], **Daniel Fišer**[3], **Isabel Valera**[1], **Verena Wolf**[1,2], **Jörg Hoffmann**[1,2]

[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2]German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
[3]Aalborg University, Denmark
{timopgros,nmueller,ivalera,wolf,hoffmann}@cs.uni-saarland.de, danfis@danfis.cz

Abstract

Recent work has shown that successful per-domain generalizing action policies can be learned. Scaling behavior, from small training instances to large test instances, is the key objective; and the use of validation instances larger than training instances is one key to achieve it. Prior work has used fixed validation sets. Here, we introduce a method generating the validation set dynamically, on the fly, increasing instance size so long as informative and feasible. We also introduce refined methodology for evaluating scaling behavior, generating test instances systematically to guarantee a given confidence in coverage performance for each instance size. In experiments, dynamic validation improves scaling behavior of GNN policies in all 9 domains used.

1 Introduction

Per-domain generalization in PDDL planning is a useful and popular setting for learning policies. Prior work has shown that successful policies of this kind can be learned using neural architectures (e.g., Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2018, 2020; Rivlin, Hazan, and Karpas 2020; Ståhlberg, Bonet, and Geffner 2022a,b, 2024; Sharma et al. 2023; Rossetti et al. 2024; Wang and Thiébaux 2024). *Scaling behavior*, the ability to generalize from small training instances to large test instances, is the key objective in this setting. Selecting the final policy based on its performance on *validation* instances, larger than the training instances, is one key to achieve that objective.

Intuitively, the larger the size difference between training and validation instances, the better policy selection can assess scaling behavior. However, prior work has relied on fixed validation sets, thereby limiting the size difference.[1] As we show here, one can instead dynamically generate larger validation instances where informative and feasible. We measure instance size in terms of the number of objects, and we fix a size-scaling scheme and random instance generator per domain. Given training instances of maximal size n_0, we generate validation instances starting at $n_0 + 1$, and we keep generating larger instances – a fixed number for each size – so long as policy coverage remains informative

(above a threshold). To ensure feasibility of this process, we impose a plan length bound.

As an additional contribution, we introduce refined methodology for rigorously evaluating scaling behavior in scientific experiments. Similarly as for validation sets, prior work has used fixed test sets, in particular ones used in International Planning Competitions (IPCs). This is, however, hardly adequate to meaningfully measure scaling behavior—consider Figure 1.
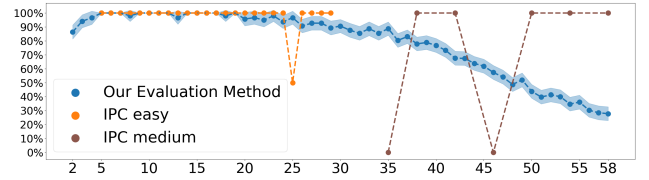

Figure 1: Scaling behavior (average coverage over instance size) of a Blocksworld policy trained following (Ståhlberg, Bonet, and Geffner 2022a), measured on IPC'23 (Taitler et al. 2024) test sets compared to our evaluation method.

Given the non-systematic nature of IPC instance sets, there are large gaps in instance size, and the average coverage per size mostly trivializes to 0% or 100%. Yet one can do much better than this, based on the same size-scaling schemes we use for validation. For each size value, we generate a sufficient number of instances to guarantee a given confidence interval. The resulting plot very clearly shows how policy performance degrades over instance size, a key insight not visible in the IPC data at all.

We also generate the training set using the same size-scaling schemes, giving much better justification to the i.i.d. assumption between training, validation, and test data.

We run experiments with graph neural network (GNN) policies trained following Ståhlberg, Bonet, and Geffner (2022a) on 9 IPC'23 domains. Dynamic validation consistently improves scaling behavior across all of these domains, with substantial advantages in 8 of them.

Our code and data are publicly available.[2]

---

[1]This also pertains to works on learning per-domain heuristic functions (e.g., Chen, Thiébaux, and Trevizan 2024); some works (Toyer et al. 2018, 2020) do not use validation at all.

[2]https://doi.org/10.5281/zenodo.15314878

2 Dynamic Validation

We here introduce our dynamic validation method. We give an overview of prior work, explain our scheme to scale instance size in a domain, 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 the best validation set performance is selected as the final policy. Algorithm 1 outlines this training loop.

Algorithm 1: Per-domain policy training loop.

Input: Training set T, validation set V, epochs E
Output: Policy π_{best}
1 $\pi_0 \leftarrow$ random ; $\pi_{\text{best}} \leftarrow \pi_0$; $v_{\text{best}} \leftarrow 0$;
2 **for** $i = 1, \ldots, E$ **do**
3 $\quad$ $\pi_i = \text{train}(\pi_{i-1}, T)$;
4 $\quad$ $v_i = \text{validate}(\pi_i, V)$;
5 $\quad$ **if** v_i better than v_{best} **then**
6 $\quad\quad$ $\pi_{\text{best}} \leftarrow \pi_i$; $v_{\text{best}} \leftarrow v_i$;

For the validation in line 4 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). This, however, limits the instances available for validation to only those that can be solved by a planner. 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 planner on the validation set (Rossetti et al. 2024).

All prior approaches to validation in per-domain policy learning, to the best of our knowledge, rely on a pre-defined 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.

Systematic instance size scaling. 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 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 constraint satisfaction problem (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 the number of children v_1, trays v_2, and sandwiches v_3. 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 $n = 3v_1 + v_2 + v_3 + 3 \land v_1 \leq v_3$.

Another example is the Rovers domain where several rovers need to navigate between waypoints to fulfill objectives, such as gathering soil data or taking images. The generator inputs are the number of rovers v_1, waypoints v_2, cameras v_3, and objectives v_4. The generator requires at least 2 waypoints, 1 lander object , 3 objects representing the camera modes, and 1 storage object for each rover. Thus, the CSP encoding is $n = 2 \cdot v_1 + v_2 + v_3 + v_4 + 4 \land 2 \leq v_2$.

For the purpose of generating an individual instance in dynamic validation, we compute a fixed number (here set to 100) of solutions to the CSP, sample one of these uniformly, and pass it as input to the generator.[3] 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.

Dynamic validation. Algorithm 2 outlines our dynamic validation method. 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. To ensure feasibility of this process, we impose a plan length bound L. The final validation score v_π of policy π is computed as the sum of achieved coverages $\mathcal{C}_i$.

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). $CSP(n, 100)$ returns 100 solutions to the CSP for size n.

---

[3]In scaling behavior evaluation (see Section 3), to be even more faithful to the generator, we compute all solutions to the CSP and sample from these uniformly.

Algorithm 2: 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 $n \leftarrow n_0$;
2 **repeat**
3     $n \leftarrow n+1$;
4   **if not** instanceOfSizeExists(n) **then**
5        $\mathcal{C}_n \leftarrow 0$; **continue** ;
6   possibleInp = $CSP(n, 100)$;
7   **for** $i \in \{1, \ldots, m\}$ **do**
8   Inp $\leftarrow$ uniform(possibleInp) ;
9        $\mathcal{I} \leftarrow$ generateInstance($\mathcal{G}$, Inp) ;
10       $\mathcal{R}_i \leftarrow$ runPolicy($\pi, \mathcal{I}, L$) ;
11    $\mathcal{C}_n \leftarrow \sum_i \mathcal{R}_i / m$;
12 **until** $\mathcal{C}_n < \tau$;
13 $v_\pi \leftarrow \sum_{i=n_0+1}^{n} \mathcal{C}_i$;

$\mathcal{R}_i$ is a Boolean whose value is 1 iff the policy found a plan. The parameters m and τ are set manually, to a fixed value used across all domains; in our experiments we use $m = 10$ and $\tau = 30\%$. To obtain the more domain-sensitive parameter L automatically, we use $L = 3N$ where N is the average length of the teacher plans on the largest training instances.

3 Scaling Behavior Evaluation

We now introduce our refined methodology for evaluating scaling behavior encapsulated in Algorithm 3.

Algorithm 3: 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 $n \leftarrow 0$; fails $\leftarrow 0$;
2 **while** fails $< \zeta$ **do**
3     $n \leftarrow n+1$; $L \leftarrow L+1$;
4   **if not** instanceOfSizeExists(n) **then continue** ;
5     $\hat{\mathcal{C}}_n \leftarrow -\infty$; $i \leftarrow 0$;
6   possibleInp = $CSP(n, \infty)$;
7   **while** $P(|\hat{\mathcal{C}}_n - \mathcal{C}_n| > \epsilon) < \kappa$ **do**
8   Inp $\leftarrow$ uniform(possibleInp) ;
9        $\mathcal{I} \leftarrow$ generateInstance($\mathcal{G}$, Inp) ;
10       $\mathcal{R}_i \leftarrow$ runPolicy($\pi, \mathcal{I}, L$) ;
11       $i \leftarrow i+1$;
12       $\hat{\mathcal{C}}_n \leftarrow \sum_{j=1}^{i} \mathcal{R}_j / i$;
13   **if** $\hat{\mathcal{C}}_n < \tau$ **then** fails $\leftarrow$ fails $+ 1$ **else** fails $\leftarrow 0$

The overall mechanics of the procedure are similar to dynamic validation. The differences are as follows. We start from $n = 1$, so as to evaluate policy performance across the entire domain. We again impose a plan length bound L on policy executions. Here, we set $L = 3N + n$, i.e., we add the current instance size, as the optimal plan length typically increases with the number of objects. 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). Instead of looking at a fixed number of instances for each size, we generate a sufficient number of instances to guarantee a given confidence interval – precisely, with parameters κ and ϵ, we follow the Chow-Robbin's method (Chow and Robbins 1965) using a Student's t-inverval (also called sequential Student's t-inverval method): we keep generating runs until, with confidence $(1-\kappa)$, the half-width of the current Student's t-interval is at most ϵ. Thus, 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.

4 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 learning track of the latest IPC in 2023 (Taitler et al. 2024). We did not add the IPC'23 domains Floortile (none of the trained GNNs could solve any instance), Spanner and Miconic (all learned policies always achieve 100% coverage, making them uninteresting for our experiments), and Sokoban (its generator throws an error in case an unsolvable instance is generated, requiring to run the generator over and over again, making it unfeasible). The generators were taken from IPC'23 where available, and otherwise from the FF domain collection.[4]

Instance sets. We use the architecture of Ståhlberg, Bonet, and Geffner (2022a). The 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). The training hyperparameters can be found in Appendix A.

We construct the training sets by uniformly sampling 100 instances per size, discarding duplicates. For the fixed validation sets, we start at size $n_0 + 1$ where n_0 is the largest training size. Per size, we generate 100 instances discarding duplicates, and uniformly select 4 of these. Full details about the training and validation sets are provided in Appendix B. We use the *seq-opt-merge-and-shrink* configuration of Fast
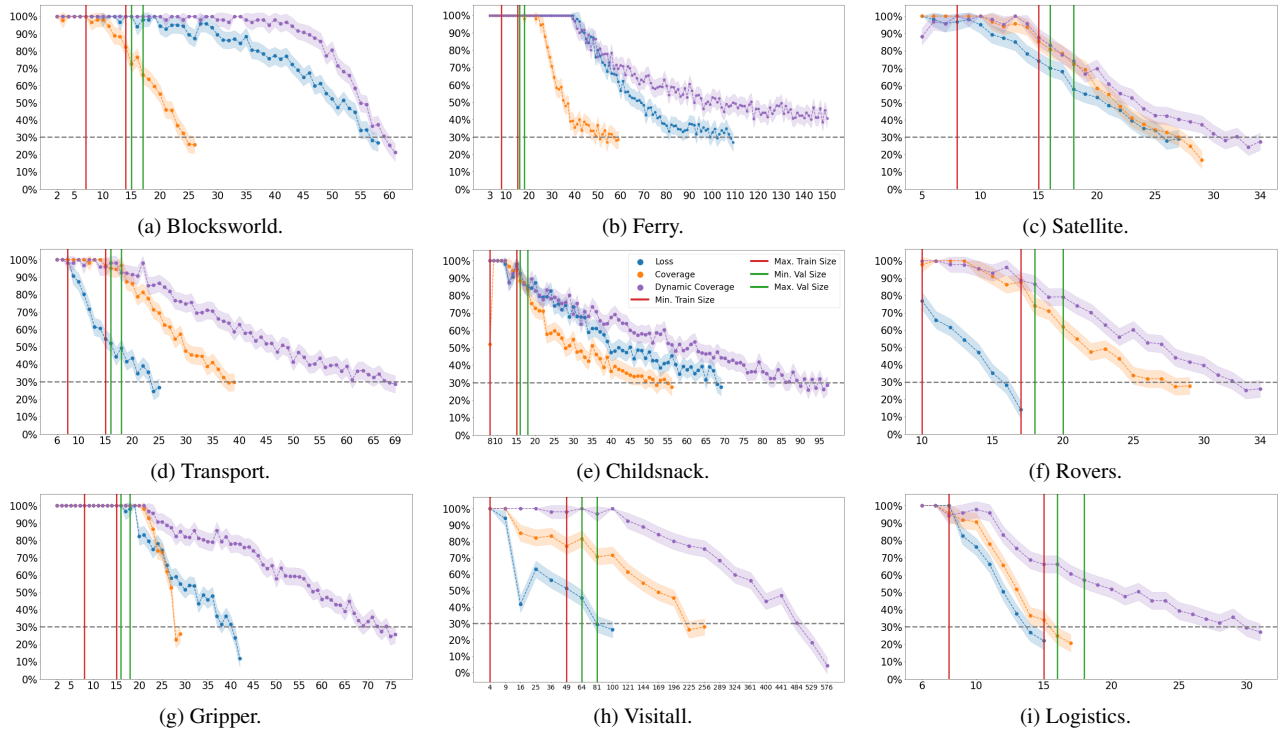
---
[4]https://fai.cs.uni-saarland.de/hoffmann/ff-domains.html

Figure 2: 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. We observed that the largest validation instance sizes seen during dynamic validation are often close to those seen in evaluation. Note that we terminated the evaluation of Ferry's dynamic coverage policy early at size 150, as $\hat{\mathcal{C}}$ stabilized above the threshold.

Downward (Helmert 2006) with limits of 20 minutes and 64 GB as the teacher planner, computing optimal plans for both sets.

Joint policy training and selection. Our setup is designed such that the training procedure is performed jointly for all 3 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 2 (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*.

In coverage, we imposed the same plan length limit as in dynamic coverage. We also imposed a one-hour time limit on the validation processes, but this was never reached in our experiments. In our experiments, dynamic coverage validation took about 8 times as long as coverage validation. However, this overhead only occurs during training.

Results. Figure 2 shows the scaling behavior evaluation of the policies validated based on loss (blue), coverage (orange), and dynamic coverage (purple).

The policies selected by the two fixed-set validation methods (loss and coverage) have mixed performances, which are similar only in the Satellite, Childsnack, and Logistics do-

	Loss		Coverage		Dynamic	
Domain	Scale	SumCov	Scale	SumCov	Scale	SumCov
Blocksworld	56	45.42	24	17.97	**59**	**52.25**
Ferry	107	72.24	57	37.74	**148**	**99.34**
Satellite	25	15.03	27	17.03	**32**	**19.33**
Transport	23	11.46	37	24.0	**67**	**41.25**
Childsnack	67	37.06	54	26.45	**95**	**51.59**
Rovers	15	3.41	27	12.57	**32**	**16.32**
Gripper	40	29.84	27	24.39	**74**	**53.98**
Visitall	64	4.81	196	9.88	**484**	**17.13**
Logistics	13	6.13	15	7.44	**29**	**15.33**

Table 1: Scale and SumCov scores of policies selected using loss, coverage, and dynamic coverage validation.

mains. Dynamic validation, however, consistently yields the best scaling behavior across all domains. Table 1 provides a summary view of these results. We measure scaling behavior here in two ways:

- *Scale* is the largest instance size n before the policy falls below the threshold τ for ζ consecutive times, measuring how far the policy can generalize with a sufficient performance; and

- *SumCov* sums up statistical coverage up to instance size n, measuring the "area below the coverage curve".
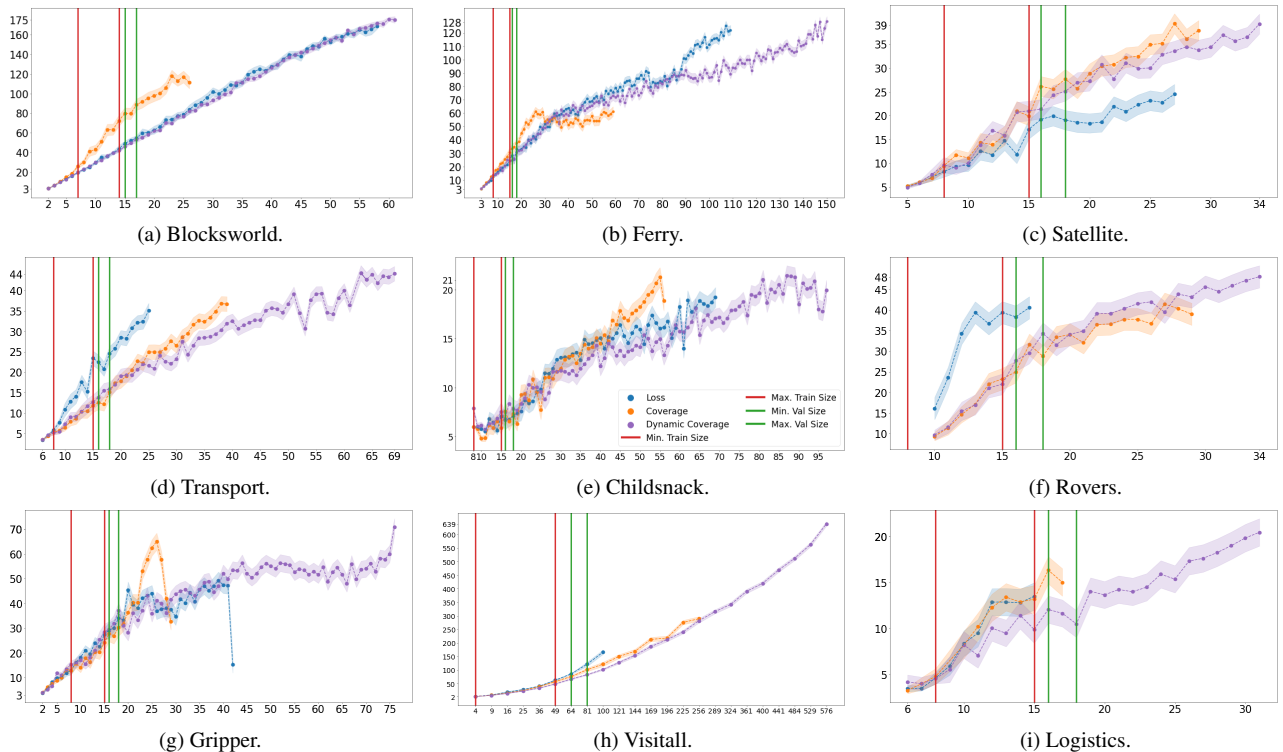
Figure 3: Evaluation of plan length using the same policies and instance sizes as in Figure 2.

Our dynamic validation method performs best in all domains in both measures: The *Scale* measure shows that the policies selected by dynamic validation generalize consistently to larger instance sizes than the policies selected by loss or coverage validation. According to the *SumCov* measure, dynamic validation policies also achieve higher total coverages across all instance sizes than either the loss or the coverage validation policies.

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

Analyzing further properties. So far, scaling behavior evaluation was only used to analyze coverage, as it is the main objective of per-domain generalizing policies. However, this method can also be applied to any other trajectory-based property. As an example, consider Figure 3, where we analyze the average plan length (by discarding runs that timeout) of the same policies and for the same instance sizes used in Figure 2.

We observe that the policies selected using dynamic coverage not only scale to larger instance sizes but additionally find plans of equal or even shorter length, with the sole exception of the Satellite domain. Given that the dynamic coverage policies solve more instances, which means also harder ones where more actions are needed, this is a very promising insight. (Another interesting comparison would be to compare the average plan length only on instances that all of the 3 approaches solve, which we leave for future work.)

5 Conclusion

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

An intriguing aspect of these improvements is that they are obtained through *policy seclection* exclusively. Perhaps there are ways to feed back insights from validation into training, guiding the training process towards better scaling behavior. We note that similar approaches have been successfully employed in the context of deep reinforcement learning (Gros et al. 2023, 2024). 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. 2020; Rossetti et al. 2024).

From the point of view of 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.

Acknowledgments

References

Chen, D. Z.; Thiébaux, S.; and Trevizan, F. 2024. Learning Domain-Independent Heuristics for Grounded and Lifted Planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 20078–20086.

Chow, Y. S.; and Robbins, H. 1965. On the Asymptotic Theory of Fixed-Width Sequential Confidence Intervals for the Mean. *The Annals of Mathematical Statistics*, 36(2): 457–462.

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.

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.

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.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 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. *Journal of Artificial Intelligence Research*, 26: 453–541.

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

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.

Sharma, V.; Arora, D.; Singla, P.; et al. 2023. SymNet 3.0: exploiting long-range influences in learning generalized neural policies for relational MDPs. In *Uncertainty in Artificial Intelligence*, 1921–1931. PMLR.

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*.

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fišer, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; Segovia-Angus, J.; and Seipp, J. 2024. The 2023 International Planning Competition. *The AI Magazine*, 45(2).

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. 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 AAAI Conference on Artificial Intelligence*.

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.

A Hyperparameters

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.

For each instance size, dynamic coverage validation generates $m = 10$ instances and stops when the coverage drops below $\tau = 30\%$. For scaling behavior evaluation, confidence intervals are computed with $\epsilon = 0.05$ and $\kappa = 0.1$, and the evaluation stops when the estimated coverage drops below $\tau = 30\%$ for $\zeta = 2$ consecutive instances.

Whereas during validation we use fixed plan length bounds L, during scaling behavior evaluation, they are scaled linearly with the instance size. The plan length bounds for each domain are listed in Table 2.

Domain	Validation	Evaluation
Blocksworld	120	$120 + n$
Ferry	78	$78 + n$
Satellite	39	$39 + n$
Transport	33	$33 + n$
Childsnack	15	$15 + n$
Rovers	60	$60 + n$
Gripper	60	$60 + n$
Visitall	114	$114 + n$
Logistics	27	$27 + n$

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

B Datasets

During plan generation, 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.

For each domain, except Visitall, we assign the instances of the eight smallest sizes to the training set. Similarly, for the validation set, 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 3.

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	$\{4, 9, 16, \ldots, 49\}$	$\{64, 81\}$
Logistics	$[8 - 15]$	$[16 - 18]$

Table 3: Number of objects in instances used for training and validation sets.