Neural Network Heuristic Functions for Classical Planning: Bootstrapping and Comparison to Other Methods

Patrick Ferber,[1, 3] **Florian Geißer,**[2] **Felipe Trevizan,**[2] **Malte Helmert,**[1] **Jörg Hoffmann**[3, 4]
[1] University of Basel, Basel, Switzerland
[2] The Australian National University, Canberra, Australia
[3] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[4] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
patrick.ferber@unibas.ch, florian.geisser@anu.edu.au, felipe.trevizan@anu.edu.au,
malte.helmert@unibas.ch, hoffmann@cs.uni-saarland.de

Abstract

How can we train neural network (NN) heuristic functions for classical planning, using only states as the NN input? Prior work addressed this question by (a) per-instance imitation learning and/or (b) per-domain learning. The former limits the approach to instances small enough for training data generation, the latter to domains where the necessary knowledge generalizes across instances. Here we explore three methods for (a) that make training data generation scalable through bootstrapping and approximate value iteration. In particular, we introduce a new bootstrapping variant that estimates search effort instead of goal distance, which as we show converges to the perfect heuristic under idealized circumstances. We empirically compare these methods to (a) and (b), aligning three different NN heuristic function learning architectures for cross-comparison in an experiment of unprecedented breadth in this context. Key lessons are that our methods and imitation learning are highly complementary; that per-instance learning often yields stronger heuristics than per-domain learning; and the LAMA planner is still dominant but our methods outperform it in one benchmark domain.

Introduction

Given the success of neural networks (NN) as game-state evaluators (Silver et al. 2016, 2017, 2018; Agostinelli et al. 2019), and the prominence of heuristic search in planning, (e.g., Hoffmann and Nebel 2001; Helmert and Domshlak 2009; Richter and Westphal 2010; Helmert et al. 2014; Domshlak, Hoffmann, and Katz 2015), NN heuristic functions are increasingly investigated.

Here, we focus on learning heuristic functions for classical planning from scratch using only states as the NN input. Prior work addressed this by (a) *per-instance* imitation learning; and/or (b) *per-domain* learning.

Ferber, Helmert, and Hoffmann (2020) and Yu, Kuroiwa, and Fukunaga (2020) use imitation learning for (a), where the NN heuristic function generalizes only over the states in the state space of the instance, and simple feed-forward NN architectures can be used. This yields NN heuristic functions competitive with the state of the art (Ferber, Helmert, and Hoffmann 2020), but it is limited to instances small enough for training data generation – solving many sample states with an off-the-shelf planner as the teacher – to be feasible.

Most works fall into category (b), exploring different variants of NN architectures based mostly on graph convolution (Garg, Bajpai, and Mausam 2019; Shen, Trevizan, and Thiébaux 2020; Rivlin, Hazan, and Karpas 2020; Karia and Srivastava 2021). Per-domain learning solves the teacher-scalability problem as it can train the NN on small instances. Yet this requires to transfer search knowledge from small instances to large ones, which is challenging and might work only for particular domains and instance distributions.

Here we explore three methods that can potentially avoid both difficulties. Two of the methods are inspired by bootstrapping (Arfaee, Zilles, and Holte 2011), where training states are generated by increasingly longer backward walks from the goal, and a search with the current NN heuristic is used to label the states. In one of these methods we estimate not goal-distance but the number of states the search needs to expand; we prove that, in an idealized setting, this converges to h^*. Our third method is based on approximate value iteration inspired by Agostinelli et al. (2019), where a k-step Bellman update is used to label the states.

We empirically compare our three methods to (a) per-instance imitation learning by Ferber, Helmert, and Hoffmann (2020), as well as (b) per-domain learning using *hypergraph networks (STRIPS-HGN)* by Shen, Trevizan, and Thiébaux (2020). Thus, we align three different NN heuristic function learning architectures for cross-comparison. Previous work has never compared NN heuristics from different works, so this is an experiment of unprecedented breadth. We believe that such cross-comparison is required to advance NN learning in planning. Finally, we compare against state-of-the-art model-based heuristics.

Key lessons from our experiments are: 1. All NN heuristic functions excel in some domains and achieve hardly anything in others. 2. The per-domain strengths of NN heuristics are highly complementary. 3. Per-instance learning often trains stronger heuristics than per-domain learning. 4. While LAMA (Richter and Westphal 2010) is generally still dominant, NN heuristic functions can outperform it in particular domains. In our experiments, this happens for the single domain (Storage) where LAMA's performance is weak.

Our appendix, code, input data, results, and the script to evaluate the results are online available (Ferber et al. 2022).

Preliminaries

We use the *FDR* planning framework (Bäckström and Nebel 1995). A planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, s_{\mathcal{I}}, \mathcal{G} \rangle$. $\mathcal{V}$ is a set of *variables*, $\mathcal{A}$ is a set of *actions*, $s_{\mathcal{I}}$ is the *initial state*, and $\mathcal{G}$ is the *goal*. Every variable has a domain $\mathcal{D}$. A *fact* is a pair $\langle v, d \rangle$ where $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$. The initial state is a complete variable assignment. The goal is a partial variable assignment. Each *action* $a \in \mathcal{A}$ defines a *precondition* pre_a and an *effect* eff_a, both are partial variable assignments. We consider unit action costs. A *plan* π is a sequence of actions $\langle a_1, \ldots, a_n \rangle$ which leads from $s_{\mathcal{I}}$ to a goal state.

Throughout this work we require some additional concepts: 1) *regressing* from a partial assignment G using an action a leads to a new partial assignment G'. Applying a on G' leads to a partial assignment $G'' \supseteq G$. 2) A *heuristic function* h estimates for every state the *cost* to the next goal state. With unit action costs, heuristics are goal-distance estimators. 3) *Bellman updates* iteratively improve state-value estimates (e.g., Bertsekas and Tsitsiklis 1996). In unit-cost classical planning, the Bellman equation simplifies to $h^*(s) = 1 + \min_{s' \in succ(s)} h^*(s')$.

Common Hyperparameters

We train NNs as heuristic functions for a given FDR task Π. Later, greedy best-first search (GBFS) uses these as heuristics. Let us first summarize the common hyperparameters.

Similar to Agostinelli et al. (2019) we use residual networks (He et al. 2016). Our NN have two dense layers, followed by one residual block with two dense layers, followed by a single output neuron. Each dense layer has 250 neurons. All neurons use the *ReLU* activation function. The NN inputs are states represented as fixed-size Boolean vectors. Every fact of Π is associated with a vector entry. If a fact is true in a state, its entry is set to 1 and to 0 otherwise. The NN outputs a single number which represents a heuristic value. We use the *mean squared error* as loss function, and the *adam* optimizer with default parameters (Kingma and Ba 2015). To prevent performance instabilities during training, we update the model for the sample generation after at least 50 epochs have passed and the loss is below 0.1. We use experience replay, i.e. the data generation pushes samples into a *first-in-first-out* buffer with a size limit of 25,000. In each epoch we uniformly choose 250 samples from the buffer.

Bootstrapping

In what follows, we introduce our three methods to train NN heuristic functions. Here, we present two methods based on *bootstrapping*. The next section introduces a method based on *approximate value iteration*.

Bootstrapping a Goal-Distance Estimator

Following Arfaee, Zilles, and Holte (2011), we train a heuristic function named h^{Boot} on states of increasing difficulty which are generated by backward walks of increasing length. We label a state s by the length of the plan obtained by GBFS on s using the current h^{Boot}.

In contrast to Arfaee, Zilles, and Holte (2011) who trained a single-layer NN to combine the estimates of model-based heuristics, we train a more complex NN to learn estimates from the FDR facts of a state. Furthermore, we apply "backward walks" in domains without fully specified goal, i.e., our goals are partial assignments. Thus, we use FDR regression. We start at the goal of Π and perform a random walk for n steps, where n is uniformly chosen from $\{0 \leq n \leq \textit{walk\_length}\}$ and *walk_length* grows iteratively. At each step, a random regressable action is chosen. The walk ends with a partial assignment. We assign each unassigned variable a random value without violating mutexes known to Fast Downward (FD, Helmert 2009).

Our other changes amount to parameter tuning. The GBFS which generates the labels has a timeout of 10 seconds. If it succeeds, we use all states s_i along the plan for training. The goal-distance estimate of s_i is the number of remaining actions on the plan. In the beginning, h^{Boot} is too uninformed to solve many states. Thus, the maximum walk length starts at 5 and doubles whenever GBFS finds a plan for more than 95% of the generated states. We double the maximum walk length at most 8 times.

Bootstrapping a Search-Space-Size Estimator

Goal distance estimates tend to correlate with search space size, but in a lose way given the highly volatile behavior of search as a function of the node ordering. Hence we introduce a variant h^{BExp} of bootstrapping, which instead estimates the search-space size of GBFS. Specifically, h^{BExp} learns to estimate the search-space size of GBFS when using h^{BExp} as the heuristic function. While this self-recursion may seem unintuitive at first, these estimates converge to h^* under idealized settings so are suitable for training a heuristic function.

The training states for h^{BExp} are labeled as follows:

$$L(s) = \begin{cases} \#\text{expansions of GBFS}(s, h^{\text{BExp}}) & \text{if } s \text{ is solvable} \\ \infty & \text{otherwise} \end{cases}$$

This assumes an idealized setting where GBFS does not operate under computational limits and hence solves all solvable states. Under the additional idealizing assumption that h^{BExp} is a look-up table rather than a function approximator, we get the stated convergence result:

Theorem 1 *If h^{BExp} uses a lookup-table $G \in \mathbb{N}^{|\mathcal{S}|}$ to store its heuristic estimates and updates the table for all states simultaneously, then it converges to h^*.*

The proof is available in the appendix. In practice, we replace the lookup table with a NN, and we enforce a time limit of 10 seconds on the GBFS. If GBFS succeeds, we use the number of expanded states as label; otherwise, we use the number of states expanded up to the time limit. The latter works better than other options (using a large constant or only the solved training states) in preliminary experiments.

Approximate Value Iteration

Our third NN heuristic function h^{AVI} is trained using approximate value iteration. Exact value iteration applies Bellman updates to a tabular value function h which maps every state s to a cost estimate. If every state s is updated in-

finitely often, h converges to h^* regardless of the update ordering (Bertsekas and Tsitsiklis 1996). Approximate value iteration replaces the value table with an approximate value function h, like an NN. This has been successfully done in single-agent puzzles including Rubik's Cube (Agostinelli et al. 2019). We adapt this approach to classical planning.

The generation of sample states s for training is done exactly as for bootstrapping, except that we keep the maximum walk length fixed. To generate the training labels, we construct the 2-step look-ahead tree of s. We evaluate all leaves as 0 if they are goal states and otherwise with h^{AVI}. Then we perform Bellman updates backwards, updating the values of intermediate states t in the tree with those of their children. The updated value at s is used for training.

Boosting NN Heuristics through Validation

In preliminary experiments, we observed that the performance of our NN heuristic functions is brittle. For a given benchmark instance, they often solve either all test states or none, with the picture changing radically after re-training. Performance is thus drastically affected by the randomness during training (e.g. parameter initialization and random walks in training data generation). As a simple remedy, we introduce a validation method. For each benchmark instance, we generate 10 new validation states. We evaluate all NN with a GBFS on the validation states with a search-time limit of 30 minutes. If less than 80% of the validation states are solved, we retrain. We retrain at most three times, and the last trained NN is used.

Experiment Methodology

We implemented our NN heuristic functions on top of FD, starting from Ferber et al.'s (2020) code base, and used Lab (Seipp et al. 2017) for our experiments. We use the Keras framework (Chollet 2015) with Tensorflow (Abadi et al. 2015) as back-end to train and evaluate our NN.

We train (including data generation) for 28 hours on 4 cores of an Intel Xeon E5-2600 processor with 4 GB memory. We use Python to update the NN and C++ to generate the training data. We test all heuristic functions in GBFS with 4 GB of memory using purely C++. We use a single core because we compare to the FF heuristic (h^{FF}, Hoffmann and Nebel 2001) and the first iteration of the LAMA planner (Richter and Westphal 2010). Both cannot exploit multiple cores. However, compared to model-based heuristics, NN heuristics profit dramatically from multiple cores or GPUs (Silver et al. 2016, 2018; Agostinelli et al. 2019). Hence, we set a generous search-time limit of 10 hours, allowing the NN heuristics to exhibit strengths in informedness.

We compare our heuristics h^{Boot}, h^{BExp}, and h^{AVI} against the imitation learning approach (h^{IL}) of Ferber, Helmert, and Hoffmann (2020), STRIPS-HGN (h^{HGN}) by Shen, Trevizan, and Thiébaux (2020), h^{FF}, and LAMA (Richter and Westphal 2010). We also run of h^{Boot}, h^{BExp}, h^{AVI}, and h^{FF} in a dual-queue search where one queue expands only the h^{FF} *preferred operators* (PO). We denote these variants by an additional $+$ in the superscript, e.g. $h^{\text{Boot+}}$.

A major part of our experimental methodology was the alignment across all the NN heuristic functions evaluated, to make the cross-comparison as fair as possible. For space reasons, we defer the full details to the appendix. One important aspect is validation (Section), which we implemented also for h^{IL} and h^{HGN}. As training data generation is independent from training for both h^{IL} and h^{HGN}, we generate the training data only once and use it to train 10 models. We reuse the validation states across methods. For h^{IL}, validation led to large coverage changes, while h^{HGN} was more robust (we observed only minor performance fluctuations).

We use the domains selected by Ferber, Helmert, and Hoffmann (2020) (cf. Table 1), and we use their instances, which were selected to be difficult enough to be interesting while easy enough to generate training data for imitation learning. Here we refer to these tasks as *moderate*, and beyond them we also consider larger *hard* instances not considered by Ferber, Helmert, and Hoffmann (2020).[1]

For each benchmark instance, we evaluate all heuristic functions on 50 distinct test states. For the moderate tasks, we use the states published by Ferber, Helmert, and Hoffmann (2020). For the hard tasks, we use their method to create test states.

Experiment Results

Table 1 summarizes our empirical findings in terms of coverage, i.e. the fraction of solved test states when using the different heuristic functions in GBFS. The table has three parts, which we will discuss in turn below. The effect of validation (cf. Section) is evaluated separately, on the moderate tasks, through the left part (w/o validation) and middle part (w/ validation) of the table. The right part of the table evaluates the heuristic functions' capability to scale to the hard tasks. Within each table part, we highlight the best-performing NN heuristic function in bold.

Validation

Consider first the data regarding validation in the left and middle part of Table 1. It shows that for each of our techniques and in most domains, validation increases coverage. The improvement is often substantial, e.g. from 31.7% to 60.3% for h^{Boot} in Depots. As validation hardly ever deteriorates performance, we keep it switched on in what follows.

Coverage Comparison for Moderate Tasks

Consider now our three techniques on the moderate tasks (cf. Table 1, middle). No method dominates the others in all domains. h^{Boot} has highest coverage in 7 domains, h^{BExp} in 4. h^{AVI} is close to the highest coverage in 2 domains.

Adding h^{IL} to the comparison, we see that it outperforms our techniques in 4 domains, h^{Boot} outperforms h^{IL} in 6 domains, h^{BExp} outperforms h^{IL} in 2 domains, and h^{AVI} outperforms h^{IL} in 2 domains. The coverage differences are dramatic in many domains.

[1] In Blocksworld and Grid, there were no larger tasks in the standard benchmarks, so we generated new ones. We do not consider the benchmarks used by Shen, Trevizan, and Thiébaux (2020), as all these instances are comparatively small.

Domain	Moderate Tasks w/o Validation			Moderate Tasks with Validation							Hard Tasks with Validation								
	h^{Boot}	h^{BExp}	h^{AVI}	h^{Boot}	h^{BExp}	h^{AVI}	h^{IL}	h^{HGN}	h^{FF}	LAMA	h^{Boot}	h^{BExp}	h^{AVI}	h^{IL}	h^{HGN}	h^{FF}	LAMA	$h^{\text{Boot+}}$	$h^{\text{FF+}}$
blocks	0	0	0	18	0	0	80	**100**	99	100	0	0	0	0	**50**	62	97	0	70
depots	32	18	**44**	60	33	55	**90**	0	98	100	8	4	13	**35**	0	36	83	24	67
grid	**100**	**100**	51	**100**	**100**	51	93	0	96	100	88	**95**	70	60	0	53	100	98	77
npuzzle	**27**	0	1	**28**	0	1	0	0	98	100	0	0	0	0	0	33	86	0	31
pipes-nt	36	**51**	21	58	68	50	**92**	8	82	99	23	19	8	**49**	0	27	69	29	64
rovers	**36**	15	34	**48**	22	45	26	14	84	100	3	1	**6**	2	0	14	100	36	96
scanaly.	33	60	**67**	33	71	67	**83**	11	98	100	3	0	**61**	60	0	98	100	8	99
storage	**89**	61	67	**89**	58	70	24	0	48	38	**27**	13	16	0	0	14	12	32	9
transport	**84**	80	70	**100**	**100**	88	99	95	98	100	0	0	**2**	0	0	0	93	0	26
visitall	**17**	0	0	55	0	0	0	**100**	93	100	28	0	0	0	**100**	74	100	32	78

Table 1: Coverage (in %). Best coverage among NN heuristic functions highlighted in **boldface** in each part of the table.

We see that h^{HGN} excels in Blocksworld and VisitAll. Here, our approaches struggle. It performs well in Transport, but fails in all other domains. Often, the reason is the hypergraph size. For many tasks in Depots, Storage and Grid, it exceeds memory. For other domains, evaluating h^{HGN} takes too much time. Blocksworld and VisitAll have very small hypergraphs: less than 1000 nodes and 1500 hyperedges.

The primary conclusion here is that *the different NN heuristic functions are highly complementary to each other*. No heuristic dominates any other, and each approach favors a subset of the domains. There seems, however, to be a tendency that *per-instance learning often yields more effective heuristics than per-domain learning by* h^{HGN}.

Turning to the comparison with model-based planners, LAMA and h^{FF} are highly competitive across all domains, generally outperforming all learning-based approaches. Indeed LAMA has perfect or almost perfect coverage everywhere, so it is impossible to beat on these benchmarks. Storage is the only domain where LAMA struggles. In that domain, all three of our approaches – but neither h^{IL} nor h^{HGN} – outperform LAMA. h^{Boot} solves almost 90% of the tasks compared to only 39% for LAMA and 48% for h^{FF}.

Coverage Comparison for Hard Tasks

Finally, consider the hard task (Table 1, right). Coverage drops for all techniques in most domains. Qualitatively though, the comparison across approaches is similar to the moderate tasks. The trained heuristic functions are now even more complementary. h^{HGN} still excels in VisitAll, but degrades heavily in Transport and Blocksworld.

Regarding h^{IL}, Ferber, Helmert, and Hoffmann (2020) did not consider the hard tasks as training data generation was expected to be a bottleneck. Indeed, this happens in some domains (e.g. Depots, NPuzzle, Rovers, and Storage). In other domains, the training data size decreases with growing instance size (to varying degrees). Yet in some cases the data is still sufficient to learn useful heuristic functions.

Our new heuristics h^{Boot}, h^{BExp}, and h^{AVI} still beat h^{FF} in Grid, and beat both h^{FF} and LAMA in Storage. Using PO improves the coverage of our heuristics and h^{FF} significantly. Our advantage in Storage increases and the margin between us and LAMA shrinks in the other domains. LAMA cannot be improved with those PO, as it already uses them. The appendix shows the improvements for h^{BExp} and h^{AVI}.

Given our comparatively large search time limit of 10 hours, we also inspected coverage over time. With few exceptions, a NN that is superior to another NN after 30 minutes remains superior across the entire time limit. On the other hand, LAMA typically solves a task either quickly or not at all, as it runs against the memory limit. The slow NN heuristics require time to "catch up", and still solve tasks after long run-times. In particular, the advantages over LAMA in Storage grow as a function of the run-time limit. Details can be found in the appendix.

Conclusion

We contribute a large experiment evaluating NN heuristic functions in classical planning, exploring three methods based on bootstrapping and approximate value iteration – one of which incorporates the new idea of estimating search effort instead of goal distance – in comparison with h^{IL} by Ferber, Helmert, and Hoffmann (2020) and h^{HGN} by Shen, Trevizan, and Thiébaux (2020). In particular, we contribute the first empirical comparison between per-domain learning and per-instance learning.

The results show that NN heuristic functions are extremely complementary, and that per-instance learning often beats per-domain learning. Our h^{Boot} heuristic outperforms both h^{IL} and h^{HGN} in 4 out of 10 domains, and in Storage even outperforms LAMA. To our knowledge, the latter is just one of two known successes of an NN heuristic function against LAMA (the other being by Karia and Srivastava (2021) on the Spanner domain from the IPC Learning track). On our other domains though, LAMA still reigns supreme.

The major open questions remain how to make the training more robust – retraining the NN can significantly change its performance – and whether and how more reliable performance can be obtained with NNs – the informedness varies wildly across domains for all evaluated methods. One may argue that this phenomenon pertains to virtually all heuristic functions, but our impression is that this is significantly more pronounced for NN than for model-based techniques.

Acknowledgments

This work was funded by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science), and by the Swiss National Science Foundation (SNSF) as part of the project "Certified Correctness and Guaranteed Performance for Domain-Independent Planning" (CCGP-Plan). Moreover, this work was partially supported by TAILOR, a project funded by the EU Horizon 2020 research and innovation programme under grant agreement no. 952215.

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Fern; a Viégas; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://tensorflow.org/.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1: 356–363.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning Heuristic Functions for Large State Spaces. *AIJ*, 175: 2075–2098.

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

Bertsekas, D. P.; and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Chollet, F. 2015. Keras. https://keras.io.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A New Systematic Approach to Partial Delete Relaxation. *AIJ*, 221: 73–114.

Ferber, P.; Geißer, F.; Trevizan, F.; Helmert, M.; and Hoffmann, J. 2022. Appendix, code, benchmarks, and experiment data for the ICAPS 2022 paper "Neural Network Heuristic Functions for Classical Planning: Bootstrapping Learning and Comparison to Other Methods". https://doi.org/10.5281/zenodo.6303621.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *Proc. ECAI 2020*, 2346–2353.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *Proc. ICAPS 2019*, 631–636.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *Proc. CVPR 2016*, 770–778.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *AIJ*, 173: 503–535.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. ICAPS 2009*, 162–169.

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

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR*, 14: 253–302.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *Proc. AAAI 2021*, 8064–8073.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *Proc. ICLR 2015*.

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

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized Planning With Deep Reinforcement Learning. In *ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*, 16–24.

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

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proc. ICAPS 2020*, 574–584.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587): 484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2017. Mastering the Game of Go Without Human Knowledge. *Nature*, 550(7676): 354–359.

Yu, L.; Kuroiwa, R.; and Fukunaga, A. 2020. Learning Search-Space Specific Heuristics Using Neural Network. In *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, 1–8.

Proof of Theorem 1

PROOF: Denote by $\mathcal{S}$ the set of all states and by H^*_n the set of states whose value is h^* after n updates: $H^*_n := \{s \mid s \in \mathcal{S}, G_n(s) = h^*(s)\}$. For convenience, we start counting the update iterations with $n = 0$. The initial lookup-table G_0 is arbitrarily initialized. We show by induction that $H^*_n \supseteq \{s | s \in \mathcal{S}, h^*(s) \leq n\}$.

Induction basis: After iteration $n = 0$, all goal states have the value 0, so, $H^*_0 \supseteq \{s \mid s \in \mathcal{S}, h^*(s) = 0\}$

Induction step: s is a state with $h^*(s) = n$. Then s has a successor s' with $h^*(s') = n - 1$. By induction hypothesis, we have $H^*_{n-1} \supseteq \{s \mid s \in \mathcal{S}, h^*(s) \leq n - 1\}$ and $s' \in H^*_{n-1}$. Thus, there is a path P from s' to the goal with G_{n-1} decreasing by 1 in each step. A GBFS run on s generates s' when expanding s, and afterwards follows P (or another path of the same length) resulting in n expansions. Hence $H^*_n \supseteq \{s | s \in \mathcal{S}, h^*(s) = n\}$. The same argument applies by induction assumption to all states t where $h^*(t) < n$: GBFS follows a direct path to the goal. So we have $t \in H^*_n$, and hence $H^*_n \supseteq \{s | s \in \mathcal{S}, h^*(s) \leq n\}$ as desired.

In all updates, GBFS run on a dead-end state s proves that s is a dead-end. Thus, all states s with infinite h^* value also satisfy $h^*(s) = G_n(s)$, for all n. This proves the claim. □

Adapting NN Training

Adaptations for $h^{\mathbf{IL}}$

Only minor changes are needed to compare with h^{IL}, which like our heuristic functions is based on per-instance learning. Ferber, Helmert, and Hoffmann (2020) generated training data for up to 400 hours, on a single CPU core. Then they trained for up to 48 hours using 4 CPU cores and 12 GB of memory. This exceeds our resource limits by far, and also Ferber et al. state themselves that, in many domains, a fraction of the training data is sufficient. Thus we adapted their resource limits to our setting, as follows.

For each benchmark instance, we generate training data on a single core for 56 hours. We train 10 heuristic functions. Each heuristic function is trained on two cores for up to 2.8 hours. Supervised learning has to keep the training data in memory, thus we use Ferber et al.'s original memory limit of 12 GB. We use validation as described above, and evaluate the resulting heuristic function h^{IL} in our experiments.

Adaptations for $h^{\mathbf{HGN}}$

STRIPS-HGN is designed for good performance with short training time, and in their original work Shen, Trevizan, and Thiébaux (2020) train the networks for only 10 minutes on small-sized problem instances. To provide a fair comparison, we adapted the training procedure of STRIPS-HGN to account for the extra training time and the source of training data used by the other learning approaches. Precisely, for each domain, we trained 10 different STRIPS-HGN networks simultaneously for up to 28 hours using 4 cores and 3.8 GB per core. We split the training time between data generation (10 hours) and network training (1.8 hour per network). Initially, we tried out different training parameters for STRIPS-HGN. We observed that, for Blocksworld, Scanalyzer and Transport, the original training time of 10 minutes and a shorter data generation time of 2 hours leads to more robust performance. We hence used this setup for these three domains. In all cases, we use the validation states to select the best STRIPS-HGN network per domain.

We generate the training data for STRIPS-HGN as follows. We sample, with replacement, a moderate or hard instance, perform the same backward walk as h^{Boot} and $h^{\text{BExp}}$for n steps (see below), and solve the generated task using A* instead of GBFS (as in the original STRIPS-HGN). We repeat this procedure until time is up. We discard every task solved within 5 minutes as they are to easy. We also discard tasks not solved after 30 minutes as it is unlikely that we will find a solution. From the solved tasks, we use the states along the (optimal) plans as training data.

The random walk length n here is uniformly chosen from $\{\underline{n} \leq n \leq \overline{n}\}$ where $\underline{n}$ and $\overline{n}$ are initially 50 and 500, respectively. Whenever our procedure generates an easy task, it updates the lower bound $\underline{n}$ to $(\underline{n} + 3n)/4$; whenever our procedure generates a timed-out task, it updates the upper bound $\overline{n}$ to $(\overline{n} + n)/2$. The number of state-value pairs obtained following this procedure ranges from 78 for Transport to 1563 for VisitAll. Al the mentioned parameters were tuned so as to optimize h^{HGN}'s performance in GBFS.

Hard Tasks with Validation

Domain	h^{Boot}	$h^{\text{Boot+}}$	h^{BExp}	$h^{\text{BExp+}}$	h^{AVI}	$h^{\text{AVI+}}$	h^{FF}	$h^{\text{FF+}}$
blocks	0	+0	0	+0	0	+0	62	+9
depots	8	+16	4	+13	13	+1	36	+31
grid	88	+11	95	+4	70	+10	53	24
npuzzle	0	+0	0	+0	0	+0	33	-2
pipes-nt	23	+6	19	+10	8	+3	27	+37
rovers	3	+34	1	+5	6	+0	14	+82
scanaly.	3	+5	0	+3	61	+6	98	+1
storage	27	+5	13	+6	16	+6	14	-5
transport	0	+0	0	+0	2	+30	0	+26
visitall	28	+4	0	+0	0	+0	74	+4

Table 2: The coverage (in %) for h^{Boot}, h^{BExp}, and h^{AVI} with and without using the preferred operators of h^{FF} on the hard tasks.

Further Results

Preferred Operators

All our techniques can be enhanced by the preferred operators of h^{FF}. For this purpose, we execute a GBFS with two open lists. The first open list uses simply the predictions of the NN, the second open list uses the same predictions, but stores only states which are reached by a preferred operator of h^{FF}. Table 2 shows that the enhanced versions dominate the base versions. For every technique, using the preferred operators leads to significant improves in multiple domains.

Coverage Over Time

Given our comparatively large search time limit of 10 hours, Figure 1 shows coverage as a function of runtime. We show results for moderate tasks in four domains; the data is qualitatively similar in the other domains and tasks.

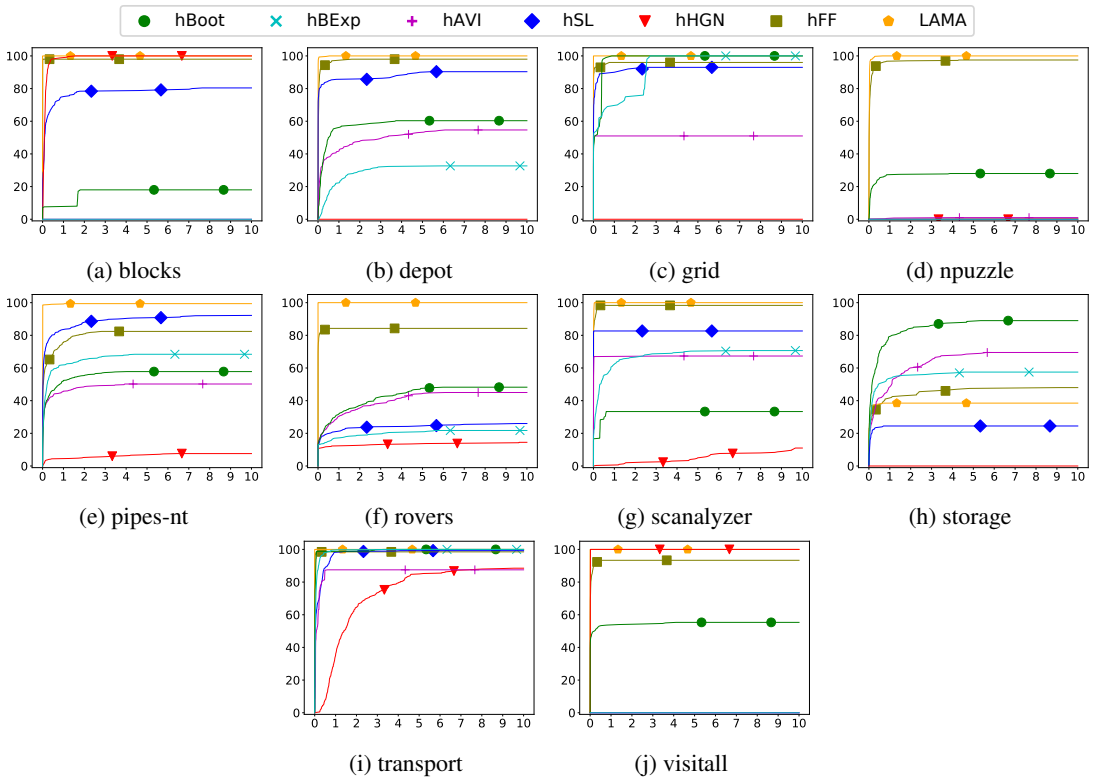


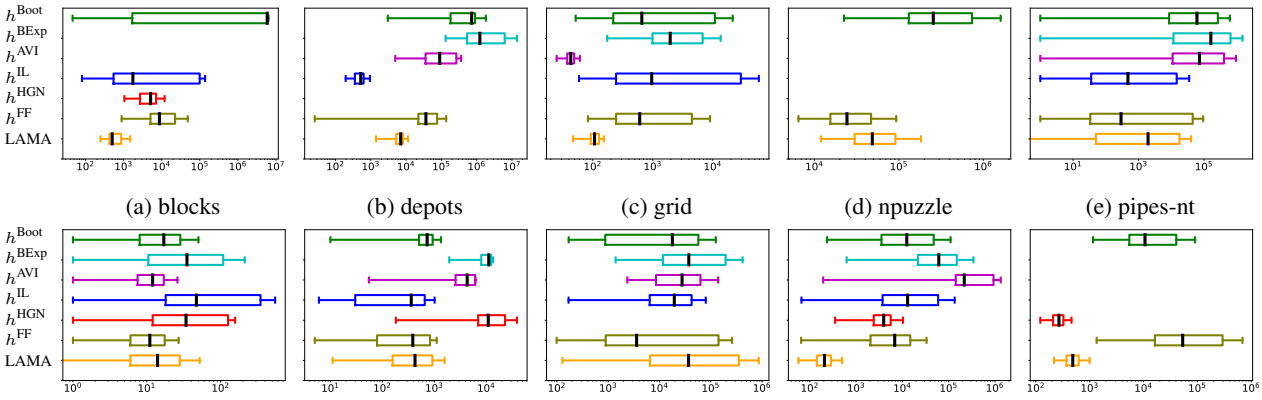
Figure 1: Coverage (%) as a function of the search time (in hours), on all moderate task for four domains.

Comparing different NN heuristics, the general finding is that coverage superiority persists over any time limit. With few exceptions, an approach that is better after 30 minutes is still better after 2 or more hours. The picture with respect to LAMA is different, as LAMA (like all state-of-the-art model-based heuristic search planners) tends to solve a task either quickly or not at all. With its comparatively fast heuristic functions, LAMA quickly runs up against the memory limit. The NN heuristic functions in contrast are very slow (run on a single core!), and thus require some time to "catch up" with LAMA. They still solve additional tasks even after very long run-times, and relative performance differences become more pronounced over time. In particular, the advantages over LAMA in Storage grow as a function of the run-time limit.

Informedness

We compare the informedness across the different approaches by comparing the number of expansions they require to solve a task. Figure 2 shows the distribution of expansions per domain, for commonly solved moderate tasks. In each domain we ignore algorithms that solve less than 10% of the tasks, as otherwise the set of commonly solved tasks would become too small.

Again, the primary conclusion from these results is that *the techniques are highly complementary* – at a glance, just consider how the different colors in Figure 2 move to and fro in the plots. Comparing neural network heuristic functions against each other, h^{HGN} is only well informed in the 3 domains in which it yields high coverage. The comparison between h^{IL} and our RL methods is similar as for coverage, exhibiting performance differences in the same domains (which is expected as the per-state runtime of these heuristic functions is very similar). Finally, the NN heuristic functions are quite competitive with h^{FF} and LAMA in terms of informedness. In Depots, Grid, and VisitAll the lowest number of expansions is achieved by a NN heuristic function (a different one in each case); and in Pipesworld-NoTankage, Rovers, and Scanalyzer, the best NN heuristic function is basically on par with h^{FF}.



(a) blocks (b) depots (c) grid (d) npuzzle (e) pipes-nt

(f) rovers (g) scanalyzer (h) storage (i) transport (j) visitall

Figure 2: Expansions on commonly solved moderate tasks, removing algorithms with coverage $< 10\%$. In each plot, the line within the body indicates the median, the body of the box plot indicates the 25 and 75 percentile, and the whiskers show the 5 and 95 percentiles.