Empirical Analysis of Abstractions for Learning Generalized Heuristic Networks

Rushang Karia and **Siddharth Srivastava**
School of Computing, Informatics and Decision Systems Engineering
Arizona State University, Tempe, AZ 85281, USA
{Rushang.Karia, siddharths}@asu.edu

Abstract

Computing goal-directed behavior is essential to designing efficient AI systems. Due to the computational complexity of planning, current approaches rely primarily upon hand-coded symbolic action models and hand-coded heuristic function generators for efficiency. Learned heuristics for such problems have been of limited utility as they are difficult to apply to problems with objects and object quantities that are significantly different from those in the training data. This paper develops a new approach for learning generalized heuristics in the absence of symbolic action models using deep neural networks that utilize an input abstraction function but are agnostic to object names and quantities. It uses an abstract state representation to facilitate data-efficient, generalizable learning. Empirical evaluation on a range of benchmark domains shows that in contrast to prior approaches, generalized heuristics computed by this method can be transferred easily to problems with different objects and with object quantities much larger than those in the training data.

1 Introduction

Given the computational complexity of automated planning [Bylander, 1991; Bylander, 1994], search-based planning algorithms often employ heuristics for efficiency [Hoffmann and Nebel, 2001; Bonet and Geffner, 2001; Helmert and Domshlak, 2009]. Designing good domain-wide heuristics as well as good domain-independent heuristic-generation principles such as "delete-relaxation" [Hoffmann and Nebel, 2001] often requires a careful study of the representation language or the structure of the underlying problems; the resulting heuristic generating functions (HGFs) are limited to planning problems where the agent's action models can be expressed using the same representation language.

This paper addresses the problem of learning domain-wide, generalizable heuristics without relying upon symbolic action models. A key requirement of the problem is that the learned heuristic be *generalizable* in the sense that it can effectively transfer to problems with different object names and/or object quantities. Recently, techniques that use deep learning to learn domain-wide [Groshev *et al.*, 2018] and domain-independent [Shen *et al.*, 2020] heuristics have demonstrated that it is possible to learn heuristics for planning.

The current landscape of learning heuristics using deep learning has two major limitations (see Sec. 6 for details). Firstly, most existing algorithms require either handwritten, symbolic action models or domain-specific network architectures. Approaches that utilize Graph Neural Networks (GNNs) (e.g., Shen *et al.* [2020]) require action models to be expressed in a representational language such as the Planning Domain Definition Language (PDDL) [Fox and Long, 2003]. Groshev *et al.* [2018] need domain-specific network architectures and input representations. Second, most approaches require large amounts of training data as input, which in turn requires good off-the-shelf planners, undermining the utility of learning heuristics in order to solve planning problems.

Recently, we introduced Generalized Heuristic Networks (GHNs) [Karia and Srivastava, 2021] as a technique for learning heuristics without requiring symbolic action models. GHNs utilize abstraction with deep learning to learn heuristic generating functions (HGFs). Our results showed that domain-wide heuristics synthesized using GHNs can efficiently generalize to problems which contain object quantities much larger than that in the training set. Furthermore, we developed leapfrogging as a few-shot meta learning technique that enables learning techniques such as GHNs to learn heuristics even in the absence of training data.

Our prior work used canonical abstractions (see Karia and Srivastava [2021]), which is a hand-coded abstraction technique. In this paper, we expand the framework of GHNs to allow for a flexible input interface that can encompass different abstraction functions including those that are learned automatically without requiring any human input. We compare obtained results with our prior work and show that GHNs can learn efficient, generalizable heuristics even when using several different abstraction functions.

The rest of this paper is organized as follows. Sec. 2 presents the necessary formal framework. Sec. 3 defines the learning problem and describes our approach for learning followed by a description of using the learned heuristic for planning (Sec. 4). Sec. 5 discusses obtained results. Sec. 6 summarizes related work followed by conclusions (Sec. 7).

2 Formal Framework

A planning *problem* is a tuple $\Gamma = \langle O, P, A, s_{init}, g, \delta \rangle$ where O is a set of objects, P is a set of predicates and A is a set of unit-cost actions. Object types can be expressed as unary predicates. The state space S for a planning problem as defined above is the set of all possible assignments of truth values to predicates in P instantiated with objects from O. $s_{init} \in S$ is the initial state and g is a goal condition expressed as a conjunctive first-order logic formula over the instantiated atoms. $\delta : S \times A \rightarrow S$ determines the transition function. Different planning problems from an application domain (e.g. Logistics) share the same P and A components and these components together define a planning *domain*, D. While a number of representations have been developed to express domain-wide, "lifted" actions [Fikes and Nilsson, 1971; Fox and Long, 2003; Sanner, 2010; Srivastava *et al.*, 2014]; such actions could also be implemented using arbitrary generative models or simulators. We assume w.l.o.g., that an action $a \in A$ can be parameterized as $a(o_1, \ldots, o_n)$ where $o_1, \ldots, o_n \in O$; we do not place any representational requirements on the specifications of A. This makes our algorithms independent of action model representations.

A solution to Γ is a plan $\pi = a_0, \ldots, a_{n-1}$ which is a sequence of actions inducing a trajectory $\tau = s_0, \ldots, s_n$ such that $s_0 \equiv s_{init}$, $\delta(s_i, a_i) = s_{i+1}$ and $s_n \models g$. The plan length $|\pi|_{s_i}$ from a state s_i is the number of states starting from s_{i+1} in τ.

A planning *heuristic* is a function $h : S \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$, where $h(s)$ estimates the cost of reaching the goal state from a state s. A heuristic is *admissible* if it never over-estimates the cost to reach the goal for any state. We define a *heuristic generating function* (HGF) as a function that maps a planning problem to a heuristic. HGFs can be domain-independent (e.g. delete-relaxation) or domain-specific. Typically, search algorithms maintain a priority queue of promising paths and use the heuristic function to compute the keys in this queue [Russell and Norvig, 2010]. For example, the priority key used in A* is $f(n) = g(n) + h(n)$ where $g(n)$ is the length of the path to a search node n and $h(n)$ is the heuristic value of the state represented in n; the node expanded is the one with the minimum value of $f(n)$ [Hart *et al.*, 1968].

We define an abstraction function, $\alpha : s \rightarrow \overline{s}$, as a function that takes as input a concrete state s and returns an abstract representation of the state $\overline{s}$. The abstraction function is many-to-one in that several concrete states can map to the same abstract state. We do not place any representational requirements on the formulation allowing our approach to be used with several different abstraction functions.

3 The Generalized Heuristic Learning Problem

We define the problem of learning generalized heuristics from training data as follows:

Definition 3.1. *(Learning Generalized Heuristics) Given a dataset of trajectories of the form $\Xi = \{\langle \pi, \tau, g, O \rangle\}$ for a domain $D = \langle P, A \rangle$ where O is a set of objects, g is a goal formula, $\tau = s_0, \ldots, s_n$, $\pi = a_0, \ldots, a_{n-1}$ contain states and parameterized actions from a planning problem $\langle D, O, s_{init}, g, \delta \rangle$ such that $s_0 \equiv s_{init}, \delta(s_i, a_i) = s_{i+1}$ and $s_n \models g$, learn a domain-wide generalized heuristic function h_D s.t. $h_D(s, g', O')$ estimates, for any planning problem $\Gamma' = \langle D, O', s'_{init}, g', \delta' \rangle$ and any state s in the state space of Γ', the distance from s to a state s' s.t. $s' \models g'$.*

In the *self-training* variant of the problem, we replace the trajectory dataset with a generator for the domain that can create problem instances with a given range of objects. Our overall approach for model-agnostic planning involves solving these learning problems by training a Generalized Heuristic Network (GHN) (Sec. 3) and using the learned GHN for planning (Sec. 4). This gives us a domain-independent method for learning domain-specific heuristic generating functions (HGFs) using either training data or problem generators. In the standard planning paradigm, GHNs would play a role similar to that of HGFs, which are currently hand-coded. Our algorithms for learning GHNs are model-agnostic in that they use only the action names and parameters, the true atoms of a state, the goal formula, and the objects in the problem, which could be provided by a blackbox simulator.

Vanilla learning for generalized heuristics To gather the training data T, we first generate a set of problem instances and use an off-the-shelf solver to compute a plan for each problem to form a library of trajectories $\Xi = \{\langle \pi, \tau, g, O \rangle\}$. Next, for each trajectory $\xi \in \Xi$, we form tuples $(s, a, |\pi|_s)$ that are then converted to $(s, \overline{s}, a, |\pi|_s)$ using an abstraction function and added to T. As a part of the data generation process, we maintain a set of features Φ, actions $\mathcal{A}$, and the maximum number of action parameters $\mathcal{A}_{max}$ that occurred in the training data. Together, they define the input-output dimensions of the network. Once T has been generated, we use standard optimization techniques to minimize the loss.

Self-training generalized heuristics using leapfrogging The training data generation method discussed above assumes access to a planner that can already solve training problems from the domain. In the absence of such a planner, we utilize leapfrogging [Groshev *et al.*, 2018] with a problem generator to interleave the learning of successively more general GHNs with the computation of training data using the GHNs being learned. Initially, problem instances with very few objects Γ'_0 are solved to generate training data T_0. These instances are small enough that blind search (without any heuristics) can be used to find solutions. We then use T_0 to learn a GHN $leap_0$. Next, $leap_0$ is used to solve larger planning problems, thereby creating training data T_1 for the next iteration, and so on. We use the problem generator to generate problem instances in batches $\Gamma'_0, \ldots, \Gamma'_i$ where problems in Γ'_i have more objects than those in Γ'_{i-1} and generate T_i by using $leap_{i-1}$ to solve $\Gamma'_0, \ldots, \Gamma'_i$. We then learn a new GHN $leap_i$ using T_i. Since GHNs learn knowledge independent of the number of objects, this iterative approach allows GHNs to effectively scale even in the absence of training datasets.

3.1 Network Architecture

We propose a general network architecture that is illustrated in Fig. 1. We use two networks, (a) the Action Network that predicts the action and its parameters, and (b) the Plan Length

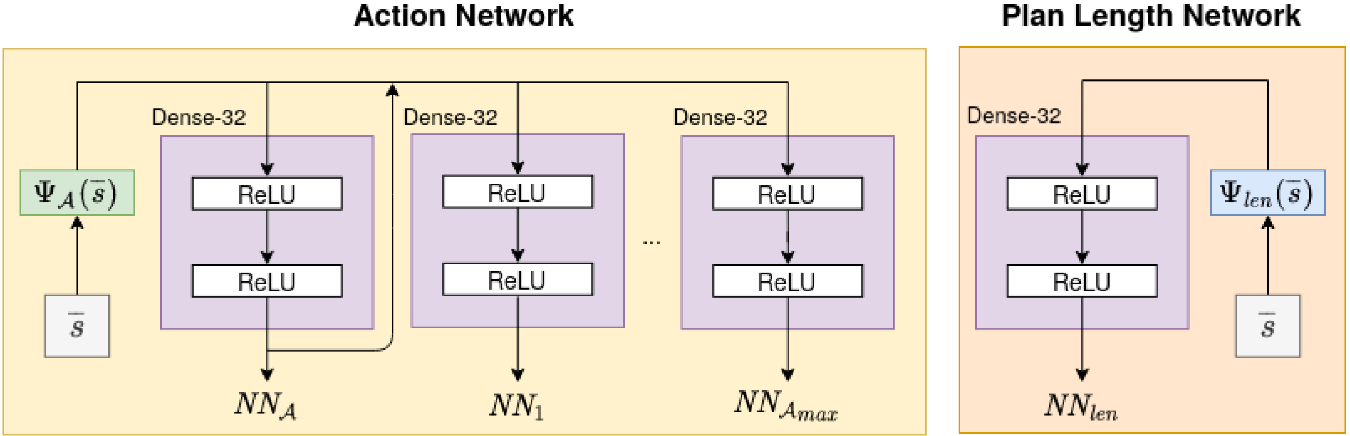

Figure 1: The network architecture used in this paper. Activations for $NN_{\mathcal{A}}$, $(NN_1, \ldots, NN_{\mathcal{A}_{max}})$ and NN_{len} are SoftMax, Sigmoid and ReLU respectively. Each *Dense-32* block contains two ReLU activated, fully-connected hidden layers with 32 tensors each. $\Psi_{\mathcal{A}}$ and Ψ_{len} are functions that convert the abstract state into vectors for the action and plan length networks respectively (described in Sec. 3.1).

Network that predicts the plan length. We found this architecture to be the most promising in our experiments.

The output of the action network is a vector $NN_{\mathcal{A}}$ of length $|\mathcal{A}|$ representing the action probability, and a set of vectors $NN_1, ..., NN_{\mathcal{A}_{max}}$ whose length is determined by the abstraction function. The output of the plan length network is a real-valued number NN_{len} that represents the predicted plan length. The action network allows for policy-based search since it allows for selecting an action whereas the plan length network can be used as a heuristic function.

The input to the neural network is an abstract state $\overline{s}$ which is converted to a vector representation by functions $\Psi_{\mathcal{A}}(\overline{s})$ and $\Psi_{len}(\overline{s})$ for the action and plan length network respectively. We assume that these functions are hand-coded based on the abstraction used. These functions along with the abstraction function as defined in Sec. 2 allow for using different abstraction functions as the input to the GHN, i.e., we are not restricted to any particular form of abstraction.

4 Planning Using Generalized Heuristic Networks

Searching using the learned heuristic network GHNs can be used in standard graph-based search algorithms like A* or GBFS using a blackbox simulator for action application and retrieving the atoms of a state. Given a node in the search tree, we use the output of the plan length network, NN_{len} to determine which node to expand next.

Using NN_{len} to compute the key in the priority queue in a search algorithm like A* or GBFS only changes the order in which the algorithm expands nodes. The actual (or real) path cost, $g(n)$ is used to determine if a visited state has been reached by a cheaper path under standard operation of the algorithm. The following result follows from the properties of such algorithms when used with a closed list [Russell and Norvig, 2010].

Theorem 4.1. *Planning with A* or GBFS using NN_{len} is sound and complete on finite state spaces.*

Hybrid heuristic function Sometimes, using the predicted path length, NN_{len} as the heuristic value in Greedy Best First Search (GBFS) can lead to poor performance since the predicted value is often approximate. To mitigate this, we propose a way to combine outputs of both the networks to form a hybrid heuristic that helps bias the search algorithm to expand promising states in the state space while adhering to the policy predicted by the network. We refer the reader to our prior work for more details.

5 Empirical Evaluation

We implemented GHN learning and tested the learned GHNs with various search algorithms (referred to as GHN/algorithm in the remainder of this section). Our implementation[1] uses Pyperplan, a popular Python-based platform for implementing and evaluating planning algorithms [Alkhazraji *et al.*, 2020].

Summary of observations Our results indicate that even though they do not use action models, (a) GHNs are competitive when compared against hand-coded HGFs, (b) in the absence of externally generated training data, leapfrogging is an effective self-training technique, (c) GHNs successfully transfer to problems with more objects than those in the training data, and (d) GHNs remain competitive even when the underlying abstraction function changes. We discuss the configuration and methods used for evaluating these hypotheses below. An extensive analysis of our results including additional problem domains is available in the appendix [Karia and Srivastava, 2020].

5.1 Empiricial Setup

We ran our experiments on Agave compute instances provided by Arizona State University. Each compute node is configured with an Intel Xeon E5-2680 v4 CPU composed of 28 cores and 128GB of RAM.

[1]Code available at https://github.com/AAIR-lab/GHN

Domain	Training Problem Parameters	Test Problem Parameters
Blocksworld	blocks $\in [2, 8]$	blocks $\in [2, 48]$
Childsnack	children, trays $\in [1, 3]$, gluten ratio=0, sandwich ratio = 1	children, trays $\in [1, 12]$, gluten ratio=0, sandwich ratio = 1
Visitall	grid dimension $\in [2, 4]$, holes $\in [0, 25]\%$, goals $\in [80, 100]\%$	grid dimension $\in [2, 12]$, holes $\in [0, 25]\%$, goals $\in [80, 100]\%$
Spanner	spanners $\in [1, 7]$, nuts $\in [1, 7]$ locations $\in [1, 7]$	spanners $\in [1, 12]$, nuts $\in [1, 12]$ locations $\in [1, 12]$
Miconic	floors $\in [2, 8]$, cars $\in [1, 8]$	locations $\in [2, 16]$, cars $\in [1, 24]$
Goldminer	rows $\in [2, 4]$, columns $\in [2, 4]$	rows $\in [2, 8]$, columns $\in [2, 8]$
Logistics	cities $\in [1, 3]$, city size=2, airplanes $\in [1, 3]$, packages $\in [1, 4]$	cities $\in [1, 4]$, city size=2, airplanes $\in [1, 5]$, packages $\in [1, 8]$
Gripper	balls $\in [1, 8]$	balls $\in [1, 32]$

Table 1: Problem generator parameters used in the generation of training and test problems.

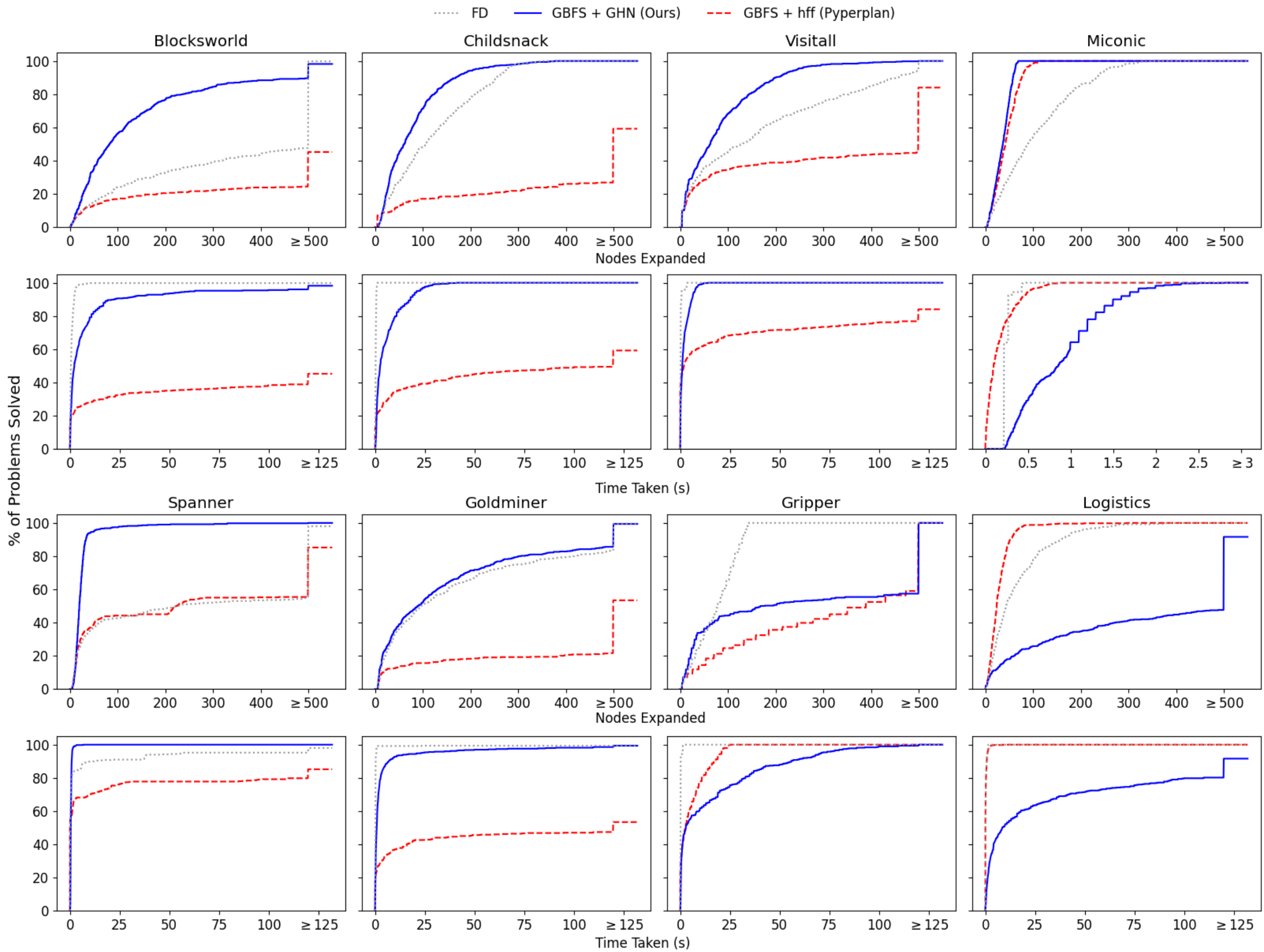

Figure 2: Performance of our approach (blue, solid) using canonical abstraction, the closest baseline implemented in an interpreted language (red, dashes) and the IPC-winning planner FD (gray, dotted). Y-axis represents the percentage of problems solved among 500 problems.

Baselines We could not find any existing domain-independent systems capable of learning HGFs without using symbolic action models. Due to the absence of suitable baselines, we compared our approach with planners and algorithms that utilize significant hand-coded, domain-specific information in the form of action models with hand-

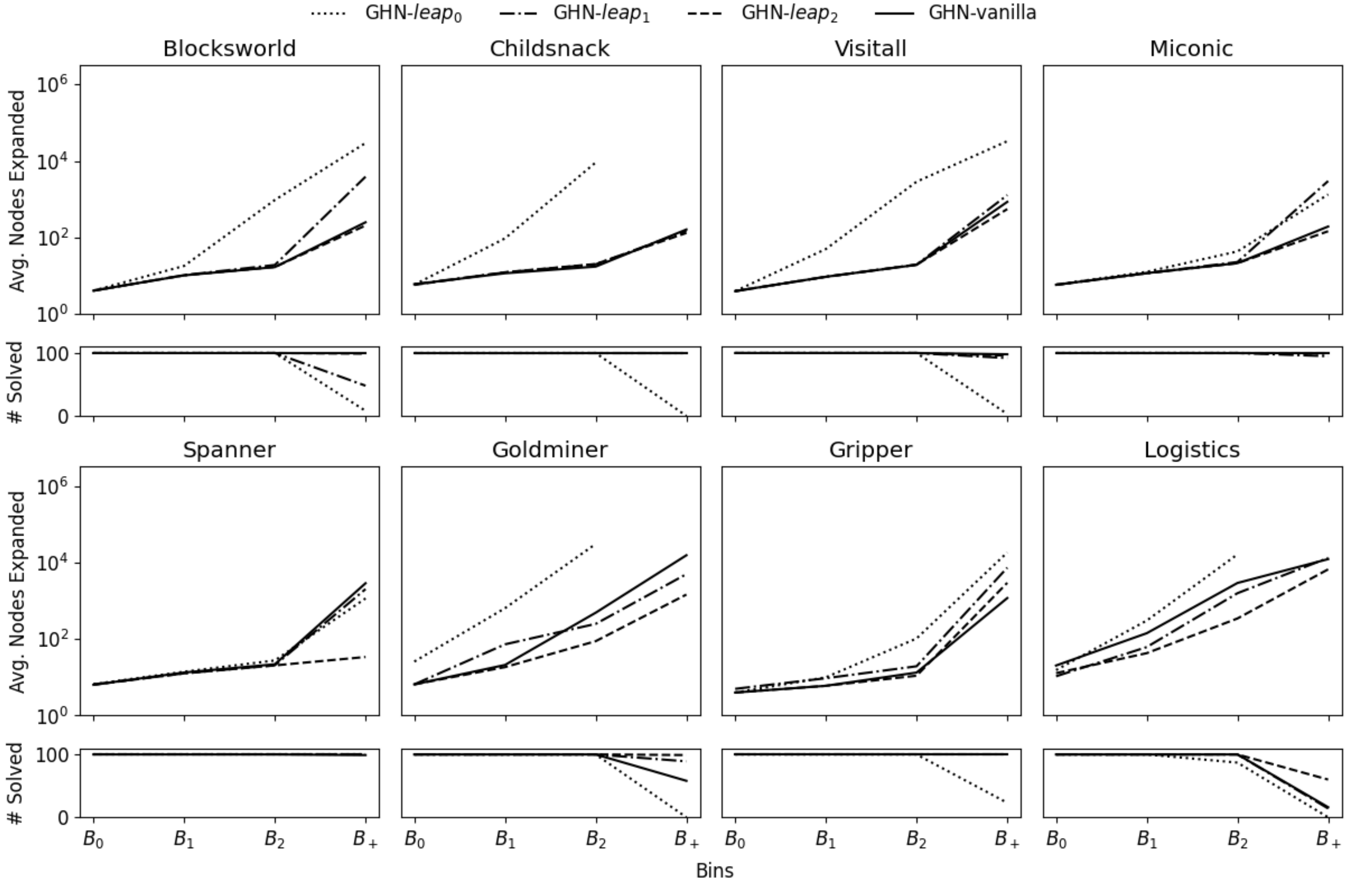

Figure 3: Performance of leapfrogging (using canonical abstraction) as a method for self-training. X-axis represents the bins for each domain. Each bin is composed of 100 test problems.

coded, domain-independent HGFs. Since such planners require domain models, we conducted an extensive evaluation using benchmarks from the International Planning Competition (IPC) [Long and Fox, 2003] that are used to evaluate such planners. While IPC winners use optimized C/C++ implementations, our approach is implemented in Python—an interpreted language that would result in slower performance than compiled languages for identical algorithms. Despite these differences in inputs and the slower performance profile of the underlying language, we found that our implementation was competitive with IPC planners.

We used 6 action-model based baselines: hand-coded HGFs {hff, lmcut} combined with search algorithms {A*, GBFS}; FF, a well-known competition winner implemented in C [Hoffmann and Nebel, 2001]; and FD LAMA, the lama-first [Richter and Westphal, 2010] configuration of Fast Downward [Helmert, 2006], also a state-of-art competition planner written in C++. hff and lmcut are implementations of the h_{FF} [Hoffmann and Nebel, 2001] and *lmcut* [Helmert and Domshlak, 2009] heuristics in Pyperplan. We denote these baselines as hff/A*, hff/GBFS, lmcut/A*, lmcut/GBFS, FF, and FD respectively. The first four baselines are implemented on the same platform (Pyperplan) as our algorithm (GHN/GBFS) and thus are particularly well-suited for comparative assessment of the strengths and weaknesses of our approach.

Abstraction functions To showcase the flexibility of the input interface of GHNs, we implemented two different abstraction functions for converting concrete states to abstract states. We used *canonical abstractions* from our prior work which is a hand-coded abstraction technique. We also present results with *qualitative features* [Bonet *et al.*, 2019; Francès *et al.*, 2021] where the abstraction function is learned automatically without requiring any human input. Please see our prior work for information about the network interface for canonical abstractions. For qualititative features, we represent $\Psi_{len}(\overline{s})$ as $\phi(s)$ and $\Psi_{\mathcal{A}}(\overline{s})$ as $[\![\phi(s)]\!]$ which converts each feature to its boolean counterpart (see Bonet *et al.*; Francès *et al.* [2019; 2021] for a formal definition). We did not utilize the action network for this abstraction function.

Test domains and problems Our evaluation consists of 13 benchmark domains from the IPC: *Blocksworld, Childsnack, Ferry*♣*, Goldminer, Grid*♣*, Gripper, Grippers*♣*, Logistics, Miconic, Sokoban*♣*, Sokoban2*♣*, Spanner,* and *Visitall*. We generated problems randomly from problem generators used by organizers of the IPC [Fawcett *et al.*, 2011]. Problem sizes were scaled by increasing the number of objects along multiple dimensions in the generator parameters. This does not necessarily increase the problem difficulty but does increase

the size of the state space. Due to space constraints, analysis for domains labeled ♣ is included in the appendix. Table 1 shows the range of generator parameters that were used for generating the training and test problems for our experiments.

Setup for self-training GHNs using leapfrogging We categorized sets of problems with increasing sizes into "bins" to showcase how leapfrogging can learn heuristics with just a problem generator in the absence of input training data. The bins were indexed as B_0, B_1, and B_2 with the number of objects monotonically increasing across several dimensions. B_+ denotes problems containing more objects than all problems in the training data. The ith leapfrog iteration, GHN-*leap*$_i$, was trained on problem sizes ranging in $B_0, \ldots, B_i$ using GHN-*leap*$_{i-1}$ to generate the training plans. Training data for GHN-*leap*$_0$ was generated using FF, however, even blind search could be used.

Training configuration We used the common network architecture paradigm illustrated in Fig. 1 to create and train all the domain-specific GHNs. Please refer to the appendix for the network hyperparameters used for training. The total training problems generated for GHN-*leap*$_0$, GHN-*leap*$_1$, GHN-*leap*$_2$ consisted of 100, 200, and 400 problems.

For our setup of vanilla GHN learning, GHN-*vanilla* used the same training problems as GHN-*leap*$_2$ but was trained directly by using FF to solve the problems and generate the training data. Each GHN used canonical abstractions (see Karia and Srivastava [2021]) for the input interface unless stated otherwise.

Test configuration To demonstrate iterative improvements in learned GHNs using leapfrogging, we used a test set of 400 problems (100 per bin) which are generated non-uniformly according to the ranges representing each bin. For example, in the Visitall domain we divided the problems based on the size n of the square grid; B_0: $n = 2$, B_1: $n = 3$, B_2: $n = 4$, B_+: $n \in \{5, \ldots, 12\}$. Bin setups for other domains can be found in the appendix.

The final leapfrog iteration, GHN-*leap*$_2$ and the baselines were run on a different test set of 500 uniformly generated problems using the parameters described in Table 1.

Evaluation metrics We focus on satisficing planning and evaluate our approach as well as the baselines on the total number of problems solved, the time taken, and the number of nodes expanded during computation.

5.2 Results and Analysis

All the baselines and GHN/GBFS (GHN-*leap*$_2$) were allocated a time limit of 600 seconds per problem. There were no restrictions on memory usage. Since no single baseline outperforms the others in every domain, we compare GHN/GBFS against the baseline configurations that outperformed their counterparts in a majority of the domains that we considered. For Pyperplan baselines this was hff/GBFS; between FD and FF, FD outperformed FF in most of the domains. Complete results for all baseline configurations are available in the appendix.

Fig. 2 and Fig. 3 summarize the key results for GHNs trained using canonical abstractions as the abstraction function. GHN/GBFS solves more problems than hff/GBFS(FD) in 6(1) of 13 domains, equal problems in 6(9) domains, and fewer problems in 1(3). When the problems solved were the same, GHN/GBFS outperformed hff/GBFS(FD) in 2(4), and underperformed on 4(5) of the domains in terms of the nodes expanded. Our analysis of the length of computed plans using GHNs indicates that GHNs are competitive with both hff/GBFS and FD and often produce cheaper plans than the baselines. Representatives of all of these categories are included in the analysis below. Our main observations are as follows:

(a) GHNs are competitive when compared against hand-coded HGFs It is clear from Fig. 2 that despite not having access to symbolic action models and hand-coded HGFs, GHNs are comparable against approaches using action models and hand-coded HGFs. Compared to Pyperplan baselines, GHNs often solve more problems and usually expend lesser effort when the number of solved problems are similar. The number of nodes expanded by GHNs is often orders of magnitude lower than the number expanded by hff/GBFS. This difference is small enough in smaller problems that the average time to solve a problem is slightly higher for GHNs due to overheads like loading the network. However, the advantages of GHNs become apparent in larger problems where GHNs can solve more problems, often requiring less time per problem despite using neural network inference to compute the heuristic value.

GHNs are also competitive when compared with FD, often expanding significantly fewer nodes and solving the same number of problems. However, despite expanding fewer nodes, GHNs are unable to compete with compiled, optimized competition planners in terms of the time taken to solve a problem. A notable exception is the Spanner domain, where FD was unable to solve many problems in B_+ and required more time to solve the problems than GHNs. The Spanner domain was specifically designed to not work well with "delete-relaxation" heuristics like those used in FD. This indicates that GHNs are able to learn knowledge of the problem structure that is orthogonal to existing heuristic generating concepts used in generating the training data.

(b) In the absence of externally generated training data, leapfrogging is an effective self-training technique Fig. 3 shows that leapfrogging is data-efficient and can learn heuristics that are comparable to, and sometimes outperform, GHN-*vanilla* which used externally generated training data. We analyze leapfrogging by considering the Visitall domain where GHN-*vanilla*, whose training data was generated using FF, is able to solve all problems in B_+. GHN-*leap*$_0$, which was the first iteration of leapfrogging was unable to solve any problems in B_+. Additionally, the performance was increasingly worse than GHN-*vanilla* on bins B_1 and B_2 indicating that the generalization capability of this iteration was limited. As the leapfrog iterations increased, the performance of the leapfrog GHNs steadily increased and the final leapfrog iteration, GHN-*leap*$_2$ was able to solve all problems in B_+, expending similar effort as GHN-*vanilla* in terms of the nodes expanded. Similar trends can be observed in other domains. This showcases leapfrogging as an effective few-shot learning technique for generating training data in a handsfree fashion.

(c) GHNs successfully transfer to problems with more objects than those in the training data As can be seen in Fig. 3, even though GHNs do not have access to action models,

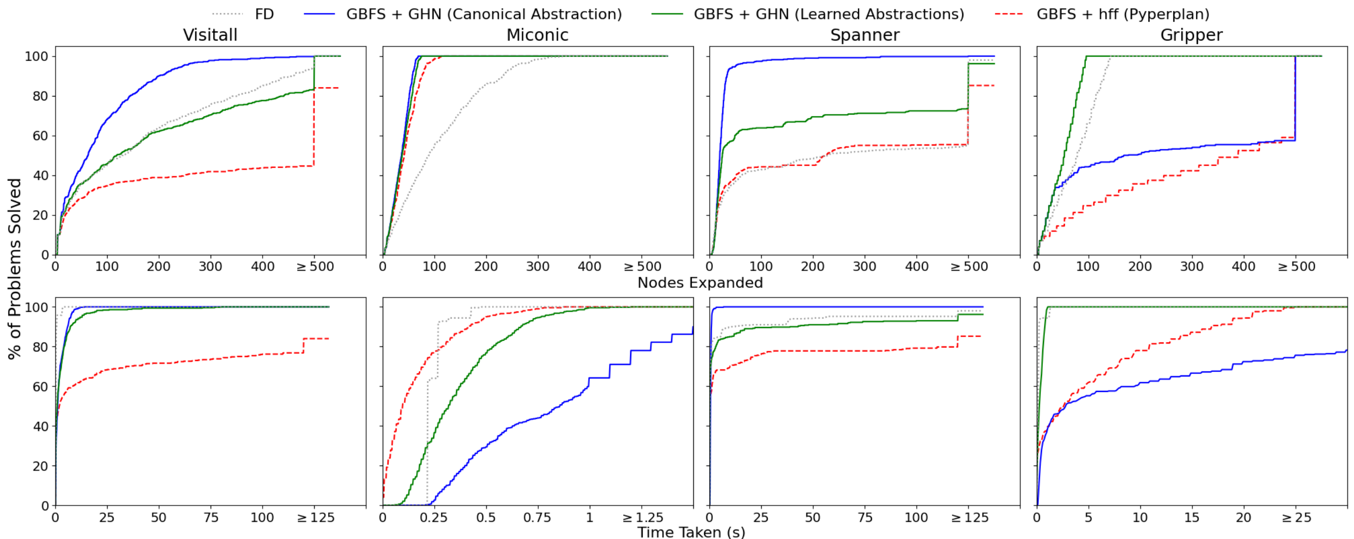

Figure 4: Performance of GHNs using abstraction features that were learned automatically (green, solid, left), GHNs using canonical abstraction (blue, solid, right), the closest baseline implemented in an interpreted language (red, dashes) and the IPC-winning planner FD (gray, dotted). Y-axis represents the percentage of problems solved among 500 problems (same set as that of Fig. 2).

GHN-*leap*$_2$ (GHN/GBFS) and GHN-*vanilla* easily transfer to problems in B_+ which consist of a greater number of objects than those in the training data. This highlights the advantages of abstraction techniques that can be used to learn HGFs that easily transfer to problems with more objects, and can be used even in the absence of action models.

GHNs appear to perform best in domains whose problems have structured solutions. We now discuss results on select domains where GHNs did not outperform the baselines. GHNs could not generalize well on the Logistics domain and were outperformed by every baseline. We investigated the reasons for the poor performance and found that one of the reasons was the nature of training data produced. The plans for Logistics are quite diverse leading to a large network loss and consequently poor search performance. One reason for this diversity could be due to the tighter coupling of objects in Logistics as is mentioned in [Rivlin *et al.*, 2020].

(d) GHNs remain competitive even when the underlying abstraction function changes. We implemented the abstraction function learned by the D2L framework [Francès *et al.*, 2021] and used it as the input interface to GHNs. Due to the difficulty in integrating the pipeline, we ran on a limited set of 4 domains. We used the same training sets, test sets and the same leapfrogging process as the GHNs that used canonical abstraction. The results (Fig. 4) show that even when using abstraction features that are automatically learned without any human-input, GHNs are able to learn heuristics that can outperform the baselines. These results show that the input interface of GHNs is flexible while retaining the ability to learn generalizable heuristics.

GHNs using automatically learned abstractions are able to outperform GHNs using canonical abstractions in domains like Gripper. We investigated the reasons and found that the abstraction features learned for Gripper captured the information of the total number of balls held by the robot. This was not accurately captured in canonical abstractions since they were not expressed by unary predicates.

On domains like Spanner, the learned abstraction performed worse than that of canonical abstraction (Fig. 2). We hypothesize that this could be due to the set of features learned being insufficient to describe spanner effectively. For example, one feature learned was one that counts the number of objects that are not held. This feature counts both the total spanners and nuts not held. We manually added another feature, *spanners-carried*(man), that counts the number of spanners carried by the agent. This change enables the GHN to beat both FD and Pyperplan baselines and performance is similar to GHNs that used hand-coded abstraction functions. However, it is interesting that even when the abstraction function learned is not ideal, GHNs are still able to outperform Pyperplan baselines.

Our results show that in a similar search setting, once the problem state spaces grow large enough, and despite using lesser information (no action models), GHNs outperform Pyperplan-based implementations, and in some cases, competition planners in the time required to solve a problem. While the computational costs of heuristic estimates using hand-coded HGFs for these problems remains fixed, the computational cost of GHNs has plenty of room for improvements. One such improvement in our implementation would be to eliminate the data structure conversion overhead that was added as a result of using FastDownward's PDDL parser instead of Pyperplan's for our internal state representation. Other optimizations such as reducing network inference costs will naturally reduce the time required to solve a problem and will bridge the gap with optimized competition planners.

6 Related Work

Our work builds upon the broad literature on learning for planning [Celorrio *et al.*, 2012; Celorrio *et al.*, 2019]. Our ap-

proach relates the most closely with other methods for learning for planning that utilize deep learning.

Value iteration networks [Tamar *et al.*, 2016] embed the standard value iteration computation within the network. While this method demonstrates successful learning, it encodes the input as an image, limiting its effectiveness in solving problems whose states do not have a natural representation as images. Groshev *et al.* [2018] learn generalized reactive policies and heuristics using a convolutional neural network (CNN). One drawback of their approach is that their network architecture and input feature vector representation are domain-dependent and require a domain expert to provide them. Feature vectors of GHNs on the other hand depend only on the abstraction function.

ASNets [Toyer *et al.*, 2018] learn generalized policies by a network composed of alternating action and proposition layers. ASNets have a fixed receptive field that can potentially limit generalizability. STRIPS-HGNs [Shen *et al.*, 2020] learn domain-independent HGFs by approximating the shortest path over the delete-relaxed hypergraph of a STRIPS [Fikes and Nilsson, 1971] problem. To do this, they define a Hypergraph Network Block, utilizing message passing to increase the receptive field of the network. The generalizability of their network depends on the number of message passing steps which can be a limiting factor as problem sizes scale up to much larger than the training data. GBFS-GNNs [Rivlin *et al.*, 2020] learn policies using network blocks similar to STRIPS-HGNs but do not use the delete-relaxed version of the problem. Since they do not learn heuristics, they use rollout during search. A common limitation of ASNets, STRIPS-HGNs, and GBFS-GNNs is that they require access to symbolic action models expressed in a language such as PDDL [Fox and Long, 2003]. While GNN-based approaches restrict states to be expressed in a relational manner, GHNs do not possess such a limitation and can be used with many different abstractions.

Curriculum learning [Bengio *et al.*, 2009] shows that effective learning is possible by organizing the training data in the form of a schedule. However, unlike leapfrogging, this method assumes that training data is available. Bootstrap learning [Arfaee *et al.*, 2010] incrementally learns a heuristic for solving a class of problems by using the heuristic learned in the current iteration to generate training data for the next iteration. However, the learned heuristic cannot generalize to problem instances with a different number of objects.

Techniques for generalized planning [Winner and Veloso, 2007; Srivastava *et al.*, 2008; Bonet *et al.*, 2009; Srivastava *et al.*, 2011; Bonet *et al.*, 2019; Francès *et al.*, 2021] primarily focus on computing algorithm-like plans and policies that can be used to solve a broad class of problems. These approaches do not generate heuristics, instead, the plan itself is computed for an arbitrary number of objects. A key limitation of these approaches compared to GHNs is that they do not provide general guarantees of completeness.

7 Conclusions

Our approach for synthesizing domain-independent HGFs differs from these prior efforts along multiple dimensions. Instead of relying on specialized network blocks, we use a rich input representation that is model-agnostic i.e. independent of action models. Using abstraction, we abstract away problem-dependent information like object names but retain the ability to capture the state structure, allowing the learned domain-wide heuristic to transfer to problems with a greater number of objects. Our empirical evaluation shows that GHNs are competitive and efficiently transfer to problems with object counts larger than those in the training data. We showed that GHNs can learn competitive heuristics even when the abstraction function changes and even if the abstraction is not ideal. Finally, in the absence of training data, we introduce leapfrogging as a few-shot learning technique that can be used to incrementally generate new training data and gradually improve the quality of the learned heuristic in a handsfree fashion.

Acknowledgements

We thank the Research Computing group at Arizona State University for providing compute hours for our experiments. This research was supported in part by the NSF under grant IIS 1942856.

References

[Alkhazraji *et al.*, 2020] Yusra Alkhazraji, Matthias Frorath, Markus Grützner, Malte Helmert, Thomas Liebetraut, Robert Mattmüller, Manuela Ortlieb, Jendrik Seipp, Tobias Springenberg, Philip Stahl, and Jan Wülfing. Pyperplan. https://doi.org/10.5281/zenodo.3700819, 2020.

[Arfaee *et al.*, 2010] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Bootstrap learning of heuristic functions. In *SOCS*, 2010.

[Bengio *et al.*, 2009] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *ICML*, 2009.

[Bonet and Geffner, 2001] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001.

[Bonet *et al.*, 2009] Blai Bonet, Héctor Palacios, and Hector Geffner. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*, 2009.

[Bonet *et al.*, 2019] Blai Bonet, Guillem Francès, and Hector Geffner. Learning features and abstract actions for computing generalized plans. In *AAAI*, 2019.

[Bylander, 1991] Tom Bylander. Complexity results for planning. In *IJCAI*, 1991.

[Bylander, 1994] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204, 1994.

[Celorrio *et al.*, 2012] Sergio Jiménez Celorrio, Tomás de la Rosa, Susana Fernández, Fernando Fernández-Rebollo, and Daniel Borrajo. A review of machine learning for automated planning. *Knowledge Eng. Review*, 27(4):433–467, 2012.

[Celorrio *et al.*, 2019] Sergio Jiménez Celorrio, Javier Segovia Aguas, and Anders Jonsson. A review of generalized planning. *Knowledge Eng. Review*, 34:e5, 2019.

[Fawcett *et al.*, 2011] Chris Fawcett, Malte Helmert, Holger Hoos, Erez Karpas, Gabriele Röger, and Jendrik Seipp. FD-Autotune: Domain-specific configuration using Fast Downward. In *ICAPS workshop*, 2011.

[Fikes and Nilsson, 1971] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *IJCAI*, 1971.

[Fox and Long, 2003] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124, 2003.

[Francès *et al.*, 2021] Guillem Francès, Blai Bonet, and Hector Geffner. Learning general policies from small examples without supervision. In *AAAI*, 2021.

[Groshev *et al.*, 2018] Edward Groshev, Maxwell Goldstein, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks. In *ICAPS*, 2018.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Helmert and Domshlak, 2009] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 2009.

[Helmert, 2006] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246, 2006.

[Hoffmann and Nebel, 2001] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.*, 14:253–302, 2001.

[Karia and Srivastava, 2020] Rushang Karia and Siddharth Srivastava. Learning generalized relational heuristic networks for model-agnostic planning (appendices). *arXiv e-prints*, 2020.

[Karia and Srivastava, 2021] Rushang Karia and Siddharth Srivastava. Learning generalized relational heuristic networks for model-agnostic planning. In *AAAI*, 2021.

[Long and Fox, 2003] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *J. Artif. Intell. Res.*, 20:1–59, 2003.

[Richter and Westphal, 2010] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res.*, 39:127–177, 2010.

[Rivlin *et al.*, 2020] Or Rivlin, Tamir Hazan, and Erez Karpas. Generalized planning with deep reinforcement learning. *arXiv e-prints*, 2020.

[Russell and Norvig, 2010] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010.

[Sanner, 2010] Scott Sanner. Relational dynamic influence diagram language (RDDL): Language description. http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf, 2010.

[Shen *et al.*, 2020] William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning domain-independent planning heuristics with hypergraph networks. In *ICAPS*, 2020.

[Srivastava *et al.*, 2008] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. Learning generalized plans using abstract counting. In *AAAI*, 2008.

[Srivastava *et al.*, 2011] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. A new representation and associated algorithms for generalized planning. *Artif. Intell.*, 175(2):615–647, 2011.

[Srivastava *et al.*, 2014] Siddharth Srivastava, Stuart J. Russell, Paul Ruan, and Xiang Cheng. First-order open-universe pomdps. In *UAI*, 2014.

[Tamar *et al.*, 2016] Aviv Tamar, Sergey Levine, Pieter Abbeel, Yi Wu, and Garrett Thomas. Value iteration networks. In *NeuRIPS*, 2016.

[Toyer *et al.*, 2018] Sam Toyer, Felipe W. Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *AAAI*, 2018.

[Winner and Veloso, 2007] Elly Zoe Winner and Manuela Veloso. Loopdistill: Learning domain-specific planners from example plans. In *ICAPS workshop*, 2007.