In Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)
Vancouver, Canada (Virtual)

Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning

Rushang Karia, 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 predicate vocabulary 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, 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, Trevizan, and Thiébaux 2020) heuristics have demonstrated that it is possible to learn heuristics for planning. However, 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, Trevizan, and Thiébaux (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.

Our approach to the problem uses abstraction with deep learning to learn heuristic generating functions (HGFs) without symbolic action models. We show that domain-wide heuristics learned using this method can efficiently generalize to problems which contain object quantities much larger than those in the training set. We also develop and evaluate leapfrogging, a bootstrapping technique that was proposed in recent work (Groshev et al. 2018) but has not been sufficiently developed and tested for learning generalized heuristics. We show that this technique facilitates *handsfree* few-shot learning (Vanschoren 2018) for competitive generalized relational heuristics without requiring external sources of training data. Meta-learning techniques like few-shot learning have seen limited applications in relational settings in prior work.

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. We will use P^k to refer to the set of predicates with arity k and P^{k+} for those with arity k or greater.

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 overestimates 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, Nilsson, and Raphael 1968).

We use *canonical abstractions* (Sagiv, Reps, and Wilhelm 2002) for representing a concrete state such that, information about object names and numbers is lifted by grouping them using abstraction predicates. Grouping together states can lead to certain predicates becoming imprecise. As a result, three-valued logic is used to represent truth values of predicates in an abstract state. We introduce canonical abstraction using the help of the following example.

Example 2.1. Consider the *gripper* domain, which consists of two rooms and a robot equipped with a pair of grippers that can pickup or drop balls (Long and Fox 2003).
Let $s_{eg} = \{\textit{free}^1(g_1), \textit{at}^2(b_1, r_a), \textit{at}^2(b_2, r_b), \textit{robotAt}^1(r_a)\}$ be a state in a *gripper* problem instance Γ expressed in *typed* PDDL with $O = \{r_a, r_b, g_1, b_1, b_2\}$, and $g = \textit{at}^2(b_1, r_b) \wedge \textit{at}^2(b_2, r_b)$. Let $\textit{type}(\textit{gripper}) = \{g_1\}$, $\textit{type}(\textit{room}) = \{r_a, r_b\}$ and $\textit{type}(\textit{ball}) = \{b_1, b_2\}$ be the object types.

Definition 2.1. *(Role) The role of an object $o \in O$ in a concrete state s is the set of unary predicates that it satisfies: $\textit{role}(o) = \{p^1 | p^1 \in P^1, p^1(o) \in s\}$.*

For the state in Example 2.1, the role of the object r_a is $\textit{role}(r_a) = \{\textit{room}, \textit{robotAt}\}$ whereas $\textit{role}(r_b) = \{\textit{room}\}$. We will use $\psi(r) = \{o | o \in O, \textit{role}(o) = r\}$ to denote the set of objects having a particular role r. Thus, $\psi(\{\textit{room}\}) = \{r_b\}$, $\psi(\{\textit{ball}\}) = \{b_1, b_2\}$, $\psi(\{\textit{room}, \textit{robotAt}\}) = \{r_a\}$ and $\psi(\{\textit{gripper}, \textit{free}\}) = \{g_1\}$. The maximum number of possible roles in a domain D with p unary predicates is 2^p.

Definition 2.2. *(Canonical Abstraction) The canonical abstraction of a concrete state $s = \{p^k(o_1, ..., o_k) | p^k \in P, o_1, ..., o_k \in O\}$ is an abstract state $\overline{s} = \{\overline{p}^k(\textit{role}(o_1), ..., \textit{role}(o_k)) | \overline{p}^k \equiv p^k\}$. Let $\mathcal{O} = \psi(\textit{role}(o_1)) \times \ldots \times \psi(\textit{role}(o_k))$ then $\overline{p}^k$ is defined as follows:*

- $\overline{p}^k(\textit{role}(o_1), \ldots, \textit{role}(o_k)) = 0 \iff \forall (o_1, \ldots, o_k) \in \mathcal{O}$ $p^k(o_1, \ldots, o_k) \notin s$.
- $\overline{p}^k(\textit{role}(o_1), \ldots, \textit{role}(o_k)) = 1 \iff \forall (o_1, \ldots, o_k) \in \mathcal{O}$ $p^k(o_1, \ldots, o_k) \in s$.
- $\overline{p}^k(\textit{role}(o_1), \ldots, \textit{role}(o_k)) = \frac{1}{2} \iff$

$$(\exists (o_1, \ldots, o_k) \in \mathcal{O} \quad p^k(o_1, \ldots, o_k) \in s) \wedge$$
$$(\exists (o_1, \ldots, o_k) \in \mathcal{O} \quad p^k(o_1, \ldots, o_k) \notin s).$$

Let $r_0 = \{\textit{gripper}, \textit{free}\}$, $r_1 = \{\textit{room}, \textit{robotAt}\}$, $r_2 = \{\textit{room}\}$ and $r_3 = \{\textit{ball}\}$ be the roles in the state s_{eg}. The canonical abstraction of the state s_{eg} is the abstract state $\overline{s_{eg}} = \{\textit{free}^1(r_0), \textit{at}^2(r_3, r_1), \textit{at}^2(r_3, r_2), \textit{robotAt}^1(r_1)\}$. The truth values for predicates in $\overline{s_{eg}}$ are $\textit{free}^1(r_0) = 1, \textit{at}^2(r_3, r_1) = \frac{1}{2}, \textit{at}^2(r_3, r_2) = \frac{1}{2}$ and $\textit{robotAt}^1(r_1) = 1$.

This formulation assumes that domains contain unary and binary predicates. Domains with ternary or higher arity predicates can be easily compiled into domains with binary predicates. The framework presented in this paper can handle higher arity predicates, however, we found that the results were best in domains compiled as binary predicates. We present a case study of Sokoban2 in the appendix (Karia and Srivastava 2020) by compiling ternary predicates present in the Sokoban domain into binary predicates.

Learning unary and binary features is an independent problem and an active area of research (Bonet, Francès, and Geffner 2019). These approaches could be used to learn such predicates for better abstractions.

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

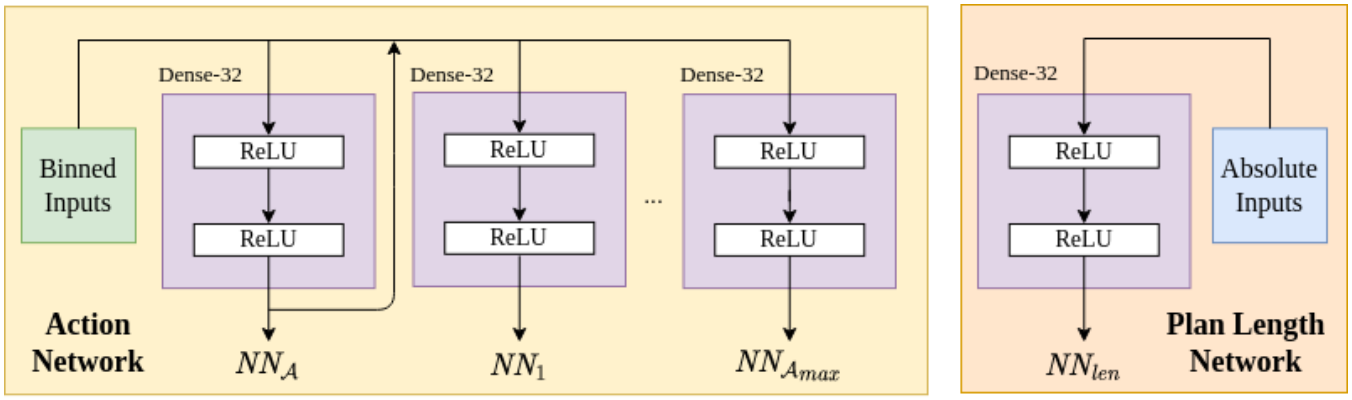

Figure 1: The network architecture used in this paper. Activations for $NN_{\mathcal{A}}$, $(NN_1, \ldots, NN_{\mathcal{A}_{max}})$ and NN_{len} are SoftMax, Sigmoid and ReLU respectively. Each *Dense-32* block contains two ReLU activated, fully-connected hidden layers with 32 tensors each. Absolute and Binned Inputs comprise vectors $v, m^2_{p_1}, \ldots, m^2_{p_n}$ and $v', m'^2_{p_1}, \ldots, m'^2_{p_n}$ respectively (described in Sec. 3.1).

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 encode goal hints to every state $s \in \tau_\xi$ using the approach in Sec. 3.2 to form tuples $(s, a, |\pi|_s)$ that are then converted to $(s, \overline{s}, a, |\pi|_s)$ using canonical abstraction (Definition 2.2) and added to T. As a part of the data generation process, we maintain a set of roles $\mathcal{R}$, actions $\mathcal{A}$, the maximum number of action parameters $\mathcal{A}_{max}$, and predicates $\mathcal{P}$ 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

The neural network used for the experiments conducted in this paper is illustrated in Fig. 1. We use two networks; one to predict the action and its parameters and the other to predict the plan length. We found this architecture to be the most promising in our experiments. We refer the reader to the appendix for our ablation study.

The output of the network is a vector $NN_{\mathcal{A}}$ of length $|\mathcal{A}|$ representing the action probability, a set of vectors $NN_1, ..., NN_{\mathcal{A}_{max}}$ each of length $|\mathcal{P}^1|$ that represents the predicted role of the corresponding parameter in the action (recall that a role is a set of unary predicates), and a real-valued number NN_{len} that represents the predicted plan length.

The input to the neural network is an abstract state that is represented as a set of vectors and matrices which capture the abstraction of object properties as well as their relationships. We compute inputs of two different types: (a) Absolute Inputs, and (b) Binned Inputs.

Absolute inputs encode the actual counts of the roles in a concrete state and also capture the role count of the k-ary atoms that are true in the state. For a concrete state s and the corresponding abstract state $\overline{s}$ we represent all roles occurring in s as a vector v of length $|\mathcal{R}|$. Each k-ary predicate $p^k \in \mathcal{P}^{2+}$ is encoded as a matrix m^k_p of dimensions $|\mathcal{R}|^k = |\mathcal{R}|_1 \times \ldots \times |\mathcal{R}|_k$. To encode absolute inputs, (a) $v[r]$ is set to the role count $|\psi(r)|$ for every role $r \in \mathcal{R}$, and (b) $m^k_p[r_i, \ldots, r_j]$ is set to the number of tuples in $\psi(r_i) \times \cdots \times \psi(r_j)$ such that $p(o_i, \ldots, o_j)$ is true in s.

Absolute inputs help in predicting the plan length since they capture information about the number of objects that belong to a particular role. However, for predicting actions, this low level of granularity is unnecessary and we found that this can lead to poor accuracy in predicting the actions. Instead, we compute binned inputs v' and ${m'}_p^k$ by categorizing the absolute inputs v and m_p^k into *levels* – which is a configurable hyperparameter – that can express information about the structure of the state at a higher level of granularity. To encode binned inputs, in our experiments, we (a) encoded $v'[i]$ as $min(v[i], 2)$ to categorize $\psi(r)$ as containing zero, one or more than one objects, and (b) encoded ${m'}_p^k[r_i, \ldots, r_j]$ as one of the three truth values of the predicate $\overline{p}^k(r_i, \ldots, r_j)$ (as defined in Definition 2.2) in $\overline{s}$. We also experimented with other strategies for encoding the binned inputs but did not observe any significant impact on the results.

3.2 Encoding Goal Hints

Inclusion of goal-relevant information has been shown to facilitate learning goal-dependent concepts (Winner and Veloso 2003; Groshev et al. 2018). We propose a simple process for encoding goal hints in concrete states just before applying state abstraction as defined in Sec. 2. This process adds new unary predicates to concrete states without using action models and takes time linear in the number of atoms in the concrete state and goal.

We first describe goal hints using an example. Consider the state s_{eg} in Example 2.1 where $g = at^2(b_1, r_b) \wedge at^2(b_2, r_b)$. We add atoms $goal^2_{at}(b_1, r_b)$ and $goal^2_{at}(b_2, r_b)$ to s_{eg}. This allows the network to identify *goal* predicates. Since $at^2(b_2, r_b) \in s_{eg}$ we also add $done^2_{at}(b_2, r_b)$ to s_{eg} which further allows the network to better identify relational structures of a state. For $at^2(b_1, r_b)$ we add two unary atoms $goal^1_{at_1}(b_1)$ and $goal^1_{at_2}(r_b)$. We similarly add two other unary atoms for $at^2(b_2, r_b)$. Doing so changes $role(r_b)$ from $\{room\}$ to $\{room, goal_{at_2}\}$ and $role(b_1)$ from $\{ball\}$ to $\{ball, goal_{at_1}\}$. These changes in object roles allow a richer representation of the abstract state since new roles demarcating objects which are part of goals have been introduced. Finally, since $at^2(b_2, r_b) \in s_{eg}$ and there is no other atom *at* appearing in the goal where b_2 is the first parameter, $done^1_{at_1}(b_2)$ is added to s_{eg} indicating that all atoms named *at* in g where b_2 appears as the first parameter are satisfied in the current state.

In general, let G refer to atoms in g for a problem Γ and let s be a concrete state. For every atom $p^k(o_1, \ldots, o_k) \in G$ we add a new atom $goal^k_p(o_1, \ldots, o_k)$ to s. This captures goal-related relational information in the state s. We also add a set of atoms $\cup_{i=1}^k \{goal^1_{p_i}(o_i)\}$ to s so that objects appearing only in G^{2+} will get a defined role in $\overline{s}$. In addition, whenever a goal atom $p^k(o_1, \ldots, o_k) \in s$, we add $done^k_p(o_1, \ldots, o_k)$ to s. Finally, if an object o appears at index i for a predicate p in s and all goal atoms named p where the object appears at index i are satisfied in s, we add $done^1_{p_i}(o)$ to s.

4 Planning Using Generalized Heuristic Networks

Hybrid heuristic function We found that 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 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 do so by evaluating a state s based on both, (1) the path from the initial state to s, and (2) the expected steps to reach the goal from s. We define the *artificial* path cost $g'(node)$ to be the sum of the action probabilities along the path from the initial state to *node.state*. Effectively, this allows us to increase the path cost of low confidence paths, which, in our experiments, enabled the search algorithm to explore promising states so that fewer nodes were expanded while computing a solution. We compute $g'(node)$ and $h(node)$ as follows:

$$V_{o+}(i, o) = \frac{\sum_{u_j \in \mathcal{P}^1 \cap role(o)} f(NN_i[u_j], \epsilon)}{|\mathcal{P}^1|} \quad (1)$$

$$V_{o-}(i, o) = \frac{\sum_{u_j \in \mathcal{P}^1 \setminus role(o)} f(1 - NN_i[u_j], \epsilon)}{|\mathcal{P}^1|} \quad (2)$$

$$V_p(i, o) = V_{o+}(i, o) + V_{o-}(i, o) \quad (3)$$

$$V_a(a(o_1, ..., o_n)) = 1 - NN_{\mathcal{A}}[a] \times \frac{\sum_{i=1}^n V_p(i, o_i)}{n} \quad (4)$$

$$g'(node) = g'(node.parent) + V_a(node.action) \quad (5)$$

$$h_{GHN}(node) = g'(node) + NN_{len} \quad (6)$$

where $role(o)$ is the role of the object o in *node.state*, $\epsilon \in [0, 1]$ is a threshold and f is a filter: $f(x, \epsilon) = 1$ if $x \geq \epsilon$ and 0 otherwise. $V_{o+}(i, o)$ and $V_{o-}(i, o)$ compute the score of the parameterized object's role relative to the predicted role. The score of the instantiated parameter o_i, $V_p \in [0, 1]$ is a ratio of the total number of unary predicates that were correctly predicted for $role(o_i)$. $V_a \in [0, 1]$ is the score of the instantiated action and can help penalize actions.

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 hybrid heuristic, h_{GHN} as described above, to determine which node to expand next.

Using h_{GHN} 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, $g(node)$ 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 h_{GHN} is sound and complete on finite state spaces.*

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

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 $\in [1, 12]$
Ferry	locations $\in [2, 4]$, cars $\in [1, 6]$	locations $\in [2, 8]$, 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 $\in [1, 8]$
Grid	x $\in [2, 5]$, y $\in [2, 5]$, key types=3, keys $\in [1, 4]$, locks $\in [1, 4]$, probability=1	x $\in [2, 8]$, y $\in [2, 8]$, key types=3, keys $\in [1, 8]$, locks $\in [1, 8]$, probability=1

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, and (c) GHNs successfully transfer to problems with more objects than those in the training data. 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.

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-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.

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 i^{th} leapfrog it-

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

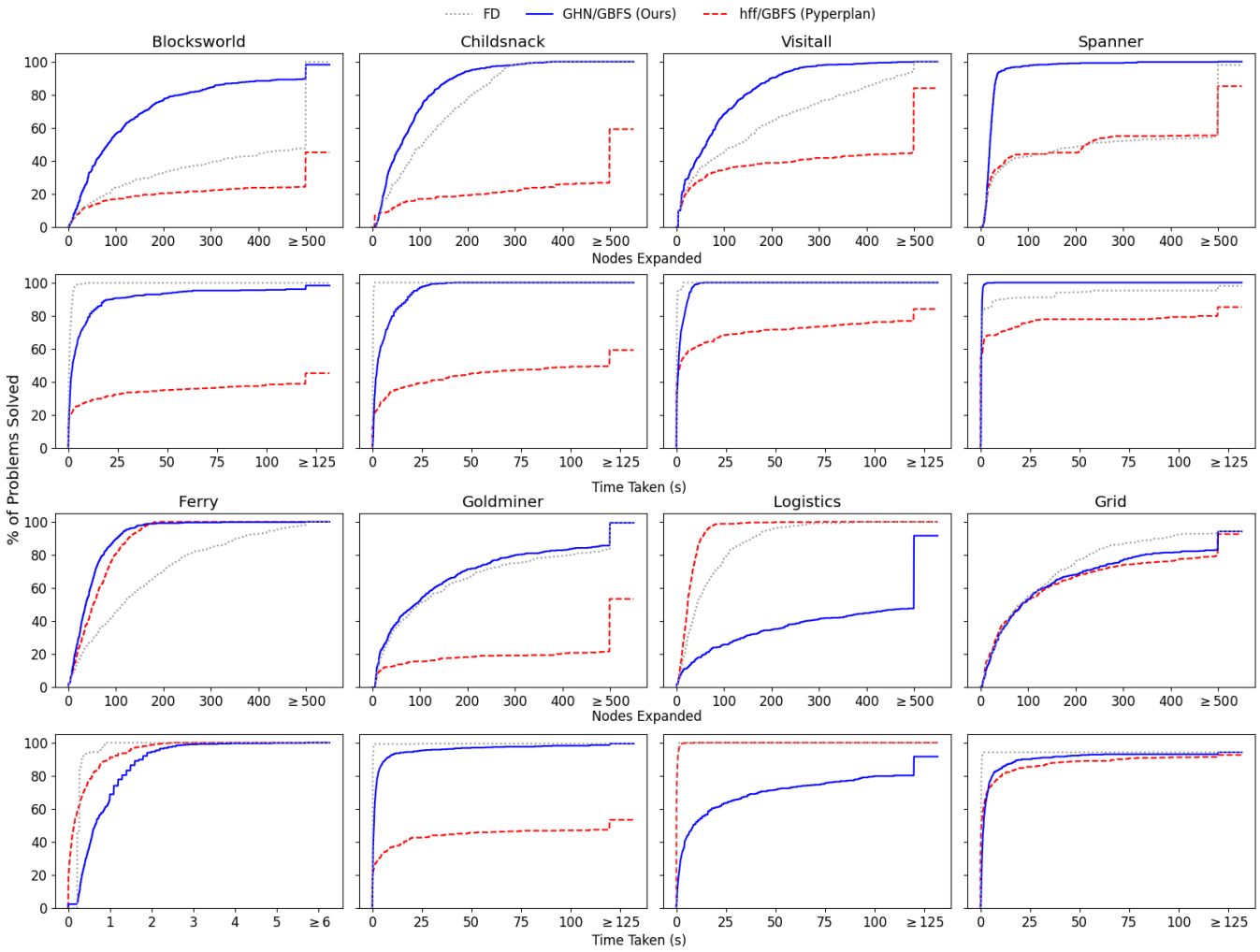

Figure 2: Performance of our approach (blue, solid), the closest baseline implemented in an interpreted language (red, dashes) and the IPC-winning planner FD (gray, dotted). Y-axis represents the percentage of problems solved among 500 problems.

eration, 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. Our optimization algorithm was the Keras (Chollet et al. 2015) implementation of RMSProp (Hinton, Srivastava, and Swersky 2012) configured with a learning rate, $\eta = 0.001$ and $\hat{\epsilon} = 1e-3$. GHNs were trained for 100 epochs using a batch size of 32. *categorical cross entropy, binary cross entropy* and *mean absolute error* were the loss minimization functions for the $NN_{\mathcal{A}}$, $NN_{1,\ldots,\mathcal{A}_{max}}$, and NN_{len} layers respectively. 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.

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 600s per problem. There were no re-

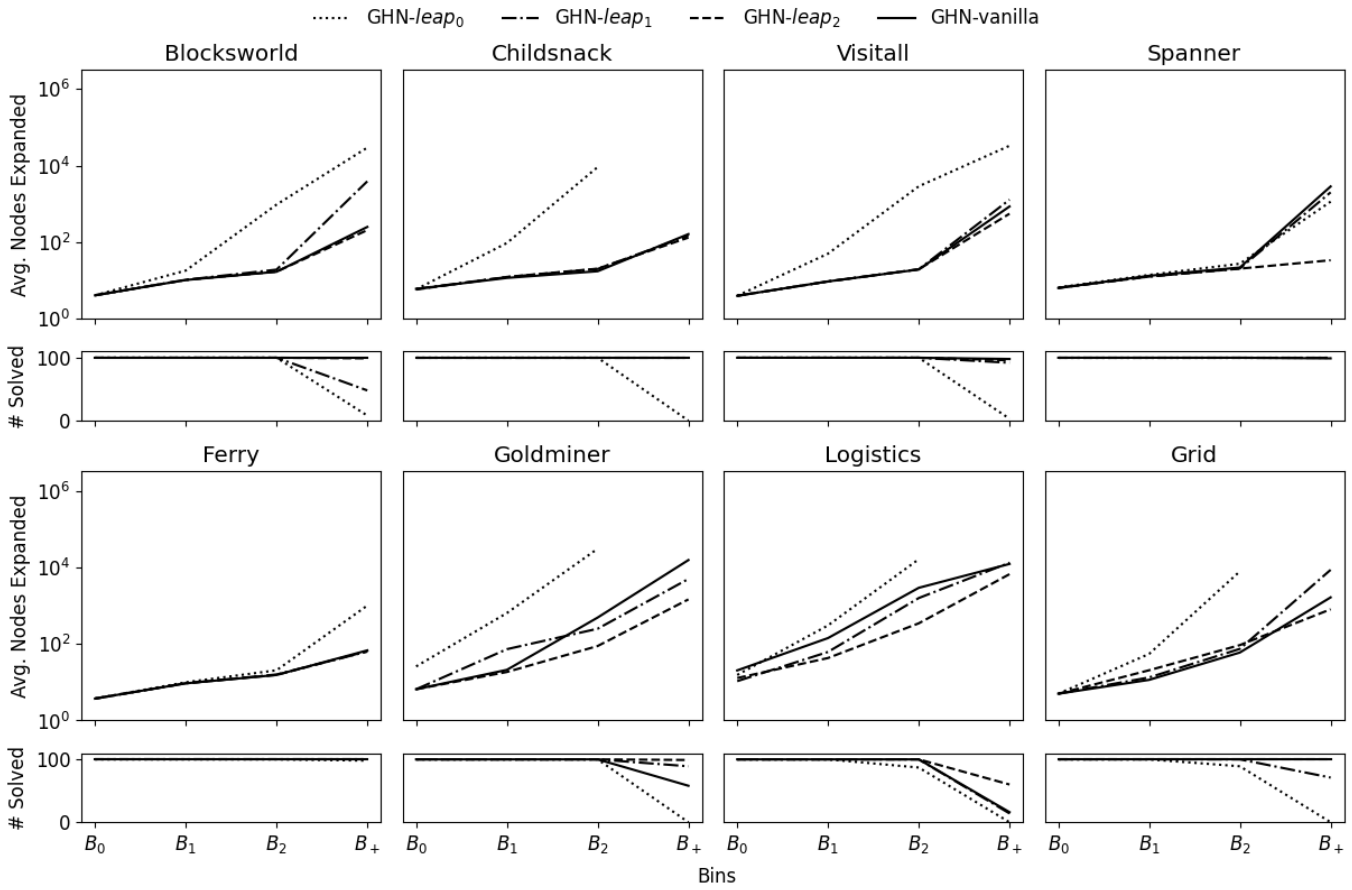

Figure 3: Performance of leapfrogging as a method for self-training. X-axis represents the bins for each domain. Each bin is composed of 100 test problems.

strictions 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 summarizes the key results. 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 Grid 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, 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, Hazan, and Karpas (2020).

Our goal encoding scheme is quite simple and cannot encode hints effectively if the goals are structured in a way that a single "goal" action provides all the goal predicates. For example, Goldminer has a very simple strategy where one needs to reach the correct y location in a grid with the right tools, and then simply move their x location to reach the goal. The goal predicate *holds-gold* is only provided in the goal state and as such the goal hints provided by our approach are not informative. Landmarks can be used to solve problems in this domain relatively easily. This information is missing in our goal encoding scheme and could be used to improve performance by learning landmarks as well. Another possible improvement would be to use first-order logic with transitive closure FO(TC) when encoding the relations so that "location" related goal information can be captured in states that are "far-away" but logically related.

We observed that GHNs have a higher network loss when actions change only binary predicates. These actions affect only the relational inputs and not the vector role counts. As a result, predictions usually have a larger error which can become quite sensitive when the number of objects is small. For example, for problems in the Spanner domain with 8 spanners but only 1 nut to tighten, GHNs had a larger test error for the predicted plan length and hence expanded more nodes. However, as the number of nuts increases, this error reduces, enabling GHNs to outperform all other baselines including FD and FF.

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 in differences with optimized competition planners.

6 Related Work

Our work builds upon the broad literature on learning for planning (Celorrio et al. 2012; Celorrio, Aguas, and Jonsson 2019). Our approach 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.

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, Trevizan, and Thiébaux 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, Hazan, and Karpas 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).

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, Zilles, and Holte 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, Immerman, and Zilberstein 2008; Bonet, Palacios, and Geffner 2009; Srivastava, Immerman, and Zilberstein 2011) primarily focus on computing algorithm-like plans 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.

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 canonical abstractions, 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. 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.

8 Ethics Statement

Automated planning is widely regarded as one of the long-standing problems of AI. This research would enable autonomous agents to carry out automated planning in the absence of domain experts. We believe that this would improve the accessibility of AI systems, as it would allow non-expert users to assign AI systems new tasks efficiently without having to invest in an AI expert who could create a symbolic domain representation and also a heuristic generating function for the task at hand.

Our approach for planning comes with guarantees of soundness and completeness. This implies that it will find a solution if there exists one, and the solution that it finds will be correct as per the simulator's action encodings. As in any approach that uses simulators, this method is susceptible to errors in programming and in simulator design. This can be addressed independently through research on formal verification of simulators used in AI.

Acknowledgements

We thank Julia Nakhleh for help with a prototype implementation of the source code. 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 grants IIS 1942856 and IIS 1909370.

References

Alkhazraji, Y.; Frorath, M.; Grützner, M.; Helmert, M.; Liebetraut, T.; Mattmüller, R.; Ortlieb, M.; Seipp, J.; Springenberg, T.; Stahl, P.; and Wülfing, J. 2020. Pyperplan. https://doi.org/10.5281/zenodo.3700819.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2010. Bootstrap Learning of Heuristic Functions. In *Proceedings of the 3rd Annual Symposium on Combinatorial Search, SOCS*.

Bengio, Y.; Louradour, J.; Collobert, R.; and Weston, J. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML*.

Bonet, B.; Francès, G.; and Geffner, H. 2019. Learning Features and Abstract Actions for Computing Generalized Plans. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence, AAAI*.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artif. Intell.* 129(1-2): 5–33.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS*.

Bylander, T. 1991. Complexity Results for Planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence, IJCAI*.

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artif. Intell.* 69(1-2): 165–204.

Celorrio, S. J.; Aguas, J. S.; and Jonsson, A. 2019. A review of generalized planning. *Knowledge Eng. Review* 34: e5.

Celorrio, S. J.; de la Rosa, T.; Fernández, S.; Fernández-Rebollo, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *Knowledge Eng. Review* 27(4): 433–467.

Chollet, F.; et al. 2015. Keras. https://keras.io.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Röger, G.; and Seipp, J. 2011. FD-Autotune: Domain-Specific Configuration using Fast Downward. In *ICAPS workshop on Planning and Learning*.

Fikes, R.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence, IJCAI*.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res.* 20: 61–124.

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*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Systems Science and Cybernetics* 4(2): 100–107.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.* 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS*.

Hinton, G.; Srivastava, N.; and Swersky, K. 2012. RMSProp. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

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

Karia, R.; and Srivastava, S. 2020. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning (Appendices). *arXiv e-prints* arXiv:2007.06702.

Long, D.; and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *J. Artif. Intell. Res.* 20: 1–59.

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

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized Planning With Deep Reinforcement Learning. *arXiv e-prints* arXiv:2005.02305.

Russell, S. J.; and Norvig, P. 2010. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education. ISBN 978-0-13-207148-2.

Sagiv, S.; Reps, T. W.; and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24(3): 217–298.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description. http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling, ICAPS*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning Generalized Plans Using Abstract Counting. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence, AAAI*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *Artif. Intell.* 175(2): 615–647.

Srivastava, S.; Russell, S. J.; Ruan, P.; and Cheng, X. 2014. First-Order Open-Universe POMDPs. In *Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence, UAI*.

Tamar, A.; Levine, S.; Abbeel, P.; Wu, Y.; and Thomas, G. 2016. Value Iteration Networks. In *Annual Conference on Neural Information Processing Systems*.

Toyer, S.; Trevizan, F. W.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies With Deep Learning. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, AAAI*.

Vanschoren, J. 2018. Meta-Learning: A Survey. *arXiv e-prints* arXiv:1810.03548.

Winner, E.; and Veloso, M. M. 2003. DISTILL: Learning Domain-Specific Planners by Example. In *Proceedings of the 20th International Conference on Machine Learning, ICML*.

Winner, E. Z.; and Veloso, M. 2007. Loopdistill: Learning domain-specific planners from example plans. In *ICAPS workshop on Planning and Scheduling*.