Learning Delete-Relaxation Heuristics over Hypergraphs

William Shen[1], **Felipe Trevizan**[2], **Sylvie Thiébaux**[2]
[1,2]The Australian National University
[1]william.w.y.shen@gmail.com [2]first.last@anu.edu.au

Abstract

We present the first approach capable of learning domain-independent planning heuristics entirely from scratch. The heuristics we learn map the hypergraph representation of the delete-relaxation of the planning problem at hand, to a cost estimate that approximates that of the least-cost path from the current state to the goal through the hypergraph. We generalise Graph Networks to obtain a new framework for learning over hypergraphs, which we specialise to learn planning heuristics by training over state/value pairs obtained from optimal cost plans. Our experiments show that the resulting architecture, STRIPS-HGNs, is capable of learning heuristics that are competitive with existing delete-relaxation heuristics including LM-cut. We show that the heuristics we learn are able to generalise across different problems and domains, including to domains that were not seen during training.

1 Introduction

Despite the prevalence of deep learning for perception tasks in computer vision and natural language processing, its application to problem solving tasks, such as planning, is still in its infancy. The majority of deep learning approaches to planning use conventional architectures designed for perception tasks, rely on hand-engineering features or encoding planning problems as images, and do not learn knowledge that generalises beyond planning with a different initial state or goal (Buffet and Aberdeen 2009; Arfaee, Zilles, and Holte 2010; Groshev et al. 2018). One exception is Action Schema Networks (ASNets) (Toyer et al. 2018; 2019), a neural network architecture which exploits the relational structure of a given planning domain described in (P)PDDL, to learn generalised policies applicable to problems of any size within the domain.

The motivation of our work is to go even further than architectures such as ASNets, and learn to plan – or at least to guide the search for a plan – independently of the domain considered. In particular, we consider the problem of learning domain-independent heuristics that generalise not only across states, goals, and object sets, but also across domains.

We focus on the well-known class of delete-relaxation heuristics for propositional STRIPS planning (Bonet and Geffner 2001; Helmert and Domshlak 2009), of which h^{max}, h^{add}, and LM-cut are popular examples. These heuristics can be seen as the least-cost path in the hypergraph representing the delete-relaxed problem for a suitable aggregation function. The vertices of this hypergraph represent the problem's propositions and the hyperedges represent actions connecting their preconditions to their positive effects. We can therefore frame the problem of learning domain-independent heuristics as that of learning a mapping from the hypergraph representation of the delete-relaxed problem (and optionally other features) to a cost estimate. To develop and evaluate this hypergraph learning framework, we make three contributions:

1. **Hypergraph Networks (HGNs)**, our novel framework which generalises Graph Networks (Battaglia et al. 2018) to hypergraphs. The HGN framework may be used to design new hypergraph deep learning models, and inherently supports combinatorial generalisation to hypergraphs with different numbers of vertices and hyperedges.
2. **STRIPS-HGNs**, an instance of a HGN which is designed to learn heuristics by approximating shortest paths over the hypergraph induced by the delete relaxation of a STRIPS problem. STRIPS-HGNs use a powerful recurrent encode-process-decode architecture which allows them to incrementally propagate messages within the hypergraph in latent space.
3. **A detailed empirical evaluation**, which rigorously defines the Hypergraph Network configurations and training procedure we use in our experiments. We train and evaluate our STRIPS-HGNs on a variety of domains and show that they are able learn domain-dependent and domain-independent heuristics which potentially outperform h^{max}, h^{add}, and LM-cut.

As far as we are aware, this is the first work to learn domain-independent heuristics completely from scratch.

2 Related Work

There is a large body of literature on learning for planning. Jimenez et al. (2012) and Toyer et al. (2019) provide excellent surveys on these existing approaches. Due to space limitations, we focus on deep learning (DL) approaches to planning which differ in what they learn, the features and architectures they use, and the generality they confer.

What is learned? Existing DL approaches may be split into four categories: learning *domain descriptions* (Say et al. 2017; Asai and Fukunaga 2018), *policies* (Buffet and Aberdeen 2009; Toyer et al. 2018; Groshev et al. 2018; Issakkimuthu, Fern, and Tadepalli 2018; Garg, Bajpai, and Mausam 2019), *heuristics* (Samadi, Felner, and Schaeffer 2008; Arfaee, Zilles, and Holte 2010; Thayer, Dionne, and Ruml 2011; Gomoluch et al. 2017), and *planner selection* (Sievers et al. 2019). Our work is concerned with learning heuristics. One of the key differences of our approach with the existing state-of-the-art for learning heuristics is that we learn heuristics from scratch instead of improving or combining existing heuristics. That being said, STRIPS-HGNs are also suitable to learn heuristic improvements or combinations, and with some adaptations, to learn actions rankings; however, we have not experimented with these settings.

Features and Architectures. Most existing DL approaches to planning use standard architectures, and rely on hand-engineered features or encodings of planning problems as images. For instance, Sievers et al. (2019) train Convolutional Neural Networks (CNNs) over graphical representations of planning problems converted into images, to determine which planner should be invoked for a planning task. For learning generalised policies and heuristics, Groshev et al. (2018) train CNNs and Graph Convolutional Networks with images obtained via a domain-specific hand-coded problem conversion. In contrast, our approach does not require hand-coded features and instead learns latent features directly from a rich hypergraph representation of the planning problem.

Another approach is ASNets (Toyer et al. 2018), a neural network architecture dedicated to planning, composed of alternating action and proposition layers which are sparsely connected according to the relational structure of the action schemas in a (P)PDDL domain. A disadvantage of ASNets is its fixed receptive field which limits its capability to support long chains of reasoning. Our STRIPS-HGNs architecture does not have such an intrinsic receptive field limitation.

Generalisation. Existing approaches and architectures for learning policies and heuristics have limited generalisation capabilities. Many generalise to problems with different initial states and goals, but not to problems with different sets or numbers of objects. Exceptions include ASNets, whose weight sharing scheme allows the generated policies to generalise to problems of any size from a given (P)PDDL domain, and TRAPSNET (Garg, Bajpai, and Mausam 2019), whose graph attention network can be transferred between different numbers of objects in an RDDL domain. As our experiments show, not only does STRIPS-HGNs support generalisation across problem sizes, but it also supports learning domain-independent heuristics that generalise across domains, including to domains that were not seen during training.

3 Planning Heuristics

We are concerned with classical planning problems represented in propositional STRIPS (Fikes and Nilsson 1971). Such a problem is a tuple $P = \langle F, O, I, G, c \rangle$ where F is the set of propositions; O is the set of actions; $I \subseteq F$ represents the initial state; $G \subseteq F$ represents the set of goal states; and $c(o)$ is the cost of action $o \in O$. Each action $o \in O$ is defined as a triple $\langle Pre(o), Add(o), Del(o) \rangle$ where the precondition $Pre(o)$ is the set of propositions which must be true in order for o to be applied, while the add- and delete-effects $Add(o)$ and $Del(o)$ are the sets of propositions which the action makes true and false, respectively, when applied.

A solution plan $\pi = o_1, \dots, o_n$ for a STRIPS problem is a sequence of applicable actions leading from the initial state to the goal, i.e., π induces a sequence of states $s_1, \dots, s_{n+1}$ such that $s_1 = I$, $G \subseteq s_{n+1}$, and for all $i \in \{1, \dots, n\}$ $s_{i+1} = (s_i \setminus Del(o_i)) \cup Add(o_i)$ and $Pre(o_i) \subseteq s_i$. The cost of a plan is the sum of the costs of its actions $\sum_{i \in \{1,\dots,n\}} c(o_i)$. An optimal plan is a plan which has minimum cost.

Heuristics. Let $\mathcal{S} \subseteq 2^F$ be the state space. A heuristic function $h \colon \mathcal{S} \to \mathbb{R}$ provides an estimate of the cost to reach a goal state from a state s, allowing a search algorithm to focus on promising parts of the state space. The optimal heuristic $h^*(s)$ is the heuristic that gives the cost of the optimal plan to reach a goal state from s. A heuristic h is admissible iff it never overestimates this optimal cost, i.e., $h(s) \leq h^*(s)\ \forall s \in \mathcal{S}$, and is inadmissible otherwise. Many heuristics are obtained by approximating the cost of the optimal plan for a relaxation of the original problem P. A well-known relaxation, the *delete-relaxation* P^+ of P is obtained by ignoring the delete-effects $Del(o)$ of all actions in P, i.e., $P^+ = \langle F, O', I, G, c \rangle$, where $O' = \{\langle Pre(o), Add(o), \emptyset \rangle \mid o \in O\}$. This works considers three baseline domain-independent heuristics which are based on the delete-relaxation: h^{max} (admissible), h^{add} (inadmissible) (Bonet and Geffner 2001), and the Landmark-Cut heuristic (admissible) (Helmert and Domshlak 2009).

4 Hypergraph Networks

Hypergraph Networks (HGNs) is our generalisation of the Graph Networks (Battaglia et al. 2018) framework to hypergraphs. HGNs may be used to represent and extend existing DL models including CNNs, graph neural networks, and state-of-the-art hypergraph neural networks. We will not explore HGNs in great detail, as it is not the focus of this paper.

Hypergraph Definition. A hypergraph is a generalisation of a graph in which a hyperedge may connect any number of vertices together. A directed hypergraph in the HGN framework is defined as a triple $G = (\mathbf{u}, V, E)$ where: $\mathbf{u}$ represents the hypergraph-level (global) features; $V = \{\mathbf{v}_i : i \in \{1, \dots, N^v\}\}$ is the set of N^v vertices where $\mathbf{v}_i$ represents the i-th vertex's features; and $E = \{(\mathbf{e}_k, R_k, S_k) \colon k \in \{1, \dots, N^e\}\}$ is the set of N^e hyperedges, where $\mathbf{e}_k$ represents the k-th hyperedge's features, R_k is the vertex set which contains the indices of the vertices which are in the head of the k-th hyperedge (i.e., receivers), and S_k is the vertex set which contains the indices of the vertices which are in the tail of the k-th hyperedge (i.e., senders). This is in contrast to Graph Networks, where R_k and S_k are singletons, i.e., $|R_k| = |S_k| = 1$. An example of a hyperedge for a delete-relaxed STRIPS action is depicted in Figure 1.

Hypergraph Network Block. A Hypergraph Network (HGN) block is a hypergraph-to-hypergraph function which forms the core building block of a HGN. The internal structure of a HGN block is identical to a Graph Network block (Battaglia et al. 2018), except now the hyperedge update function ϕ^e supports multiple receivers and senders. A *full* HGN

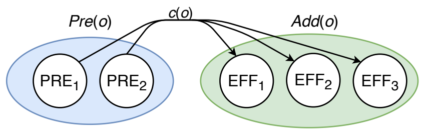

Figure 1: Our formulation of a hyperedge for an action $o \in O$ with 2 preconditions and 3 positive effects. The preconditions are the 'senders', while the positive effects are the 'receivers'.

block is composed of three update functions, ϕ^e, ϕ^v and ϕ^u, and three aggregation functions, $\rho^{e\to v}$, $\rho^{e\to u}$ and $\rho^{v\to u}$:

$$\begin{aligned} \mathbf{e}'_k &= \phi^e(\mathbf{e}_k, \mathbf{R}_k, \mathbf{S}_k, \mathbf{u}) & \overline{\mathbf{e}}'_i &= \rho^{e\to v}(E'_i) \\ \mathbf{v}'_i &= \phi^v(\overline{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}) & \overline{\mathbf{e}}' &= \rho^{e\to u}(E') \\ \mathbf{u}' &= \phi^u(\overline{\mathbf{e}}', \overline{\mathbf{v}}', \mathbf{u}) & \overline{\mathbf{v}}' &= \rho^{v\to u}(V') \end{aligned}$$

where $\mathbf{R}_k = \{\mathbf{v}_j : j \in R_k\}$ and $\mathbf{S}_k = \{\mathbf{v}_j : j \in S_k\}$ are the sets which represent the vertex features of the receivers and senders of the k-th hyperedge, respectively. Additionally, for the i-th vertex, we define $E'_i = \{(\mathbf{e}'_k, R_k, S_k) : k \in \{1, \ldots, N^e\}$ s.t. $i \in R_k\}$, $V' = \{\mathbf{v}'_i : i \in \{1, \ldots, N^e\}\}$, and $E' = \bigcup_i E'_i = \{(\mathbf{e}'_k, R_k, S_k) : k \in \{1, \ldots, N^e\}\}$. Essentially, E'_i represents the updated hyperedges where the i-th vertex is a receiver vertex, E' represents all the updated hyperedges, and V' represents all the updated vertices.

Since the input to the aggregation functions are essentially sets, each ρ must be permutation invariant to ensure that all permutations of the input give the same aggregated result. Hence ρ could, for example, be a function that takes an element-wise summation of the input, maximum, minimum, mean, etc (Battaglia et al. 2018).

Computation Steps. In a single forward pass of a HGN block, the hyperedge update function ϕ^e is firstly applied to all hyperedges to compute per-hyperedge updates. Each updated hyperedge feature $\mathbf{e}'_k$ is computed using the current hyperedge's feature $\mathbf{e}_k$, the features of the receiver and sender vertices $\mathbf{R}_k$ and $\mathbf{S}_k$, respectively, and the global features $\mathbf{u}$. Next, the vertex update function ϕ^v is applied to all vertices to compute per-vertex updates. Each updated vertex feature $\mathbf{v}'_i$ is computed using the aggregated information $\overline{\mathbf{e}}'_i$ from all the hyperedges the vertex 'receives' a signal from (i.e., it appears in the head of the hyperedge), the current vertex's feature $\mathbf{v}_i$, and the global features $\mathbf{u}$. Finally, the global update function ϕ^u is applied to compute the new global features (Battaglia et al. 2018) using the aggregated information from all the hyperedges and vertices in the hypergraph $\overline{\mathbf{e}}'$ and $\overline{\mathbf{v}}'$, respectively, along with the current global features $\mathbf{u}$.

Configuring HGN Blocks. Each update function ϕ in a HGN block must be implemented by some function f, where the signature of f determines what input it gets (Battaglia et al. 2018). For example, the function that implements ϕ^e in a full HGN block (Figure 2) is a function $f : (\mathbf{e}_k, \mathbf{R}_k, \mathbf{S}_k, \mathbf{u}) \mapsto \mathbf{e}'_k$ which accepts the global, vertex, and hyperedge attributes. Each function f may be implemented in any manner, as long as it accepts the input parameters and conforms to the required output. Since HGN blocks are hypergraph-to-hypergraph functions, we may compose blocks sequentially and repeatedly apply them.

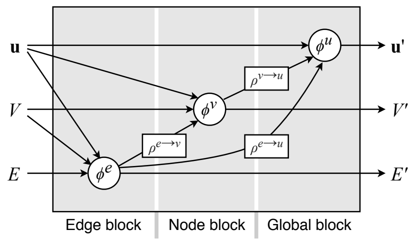

Figure 2: The full HGN block configuration which predicts global, vertex and hyperedge outputs based on the incoming global, vertex and hyperedge attributes (Figure 4a from Battaglia et al. 2018). The incoming arrows to an update function ϕ represent the inputs it receives.

5 STRIPS-HGNs

STRIPS-HGNs is our instantiation of a HGN which uses a recurrent *encode-process-decode* architecture (Battaglia et al. 2018) for learning heuristics. STRIPS-HGNs are designed to be highly adaptable to different input features for each proposition and action, as well as being agnostic to the implementation of each update function in each HGN block.

Hypergraph Representation. The input to a STRIPS-HGN is a hypergraph $G_{\text{inp}} = (\mathbf{u}_{\text{inp}}, V_{\text{inp}}, E_{\text{inp}})$ which contains the input proposition and action features for the state s, along with the hypergraph structure of the relaxed STRIPS problem $P^+ = \langle F, O', I, G, c\rangle$, where:

- $\mathbf{u}_{\text{inp}} = \emptyset$, as global features are not required as input to a STRIPS-HGN. Nevertheless, it is easy to adapt STRIPS-HGNs to support global features, e.g., we could supplement a STRIPS-HGN with a heuristic value $h(s)$ computed by another heuristic h such that the network learns an "improvement" on h similar to (Gomoluch et al. 2017).
- $V_{\text{inp}} = \{\mathbf{v}_i : i \in \{1, \ldots, |F|\}\}$ contains the input features for the $|F|$ propositions in the problem. Features for a proposition could include whether it is true for the current state or goal state, and whether the proposition is a *fact landmark* for the state s (Richter and Westphal 2010).
- $E_{\text{inp}} = \{(\mathbf{e}_k, R_k, S_k) : k \in \{1, \ldots, |O'|\}\}$ for the $|O'|$ actions in the relaxed problem P^+. For an action $o \in O'$ represented by the k-th hyperedge, $\mathbf{e}_k$ represents the input features for o (e.g., the cost of the action $c(o)$, and whether the action is in the *disjunctive action landmarks* from state s) and $R_k = Add(o)$ (resp. $S_k = Pre(o)$) is the vertex set containing the indices of the vertices in the additive effects (resp. preconditions) of o.

The output of a STRIPS-HGN is a hypergraph $G_{\text{out}} = (\mathbf{u}_{\text{out}}, V_{\text{out}}, E_{\text{out}})$ where $\mathbf{u}_{\text{out}} \in \mathbb{R}^{1\times 1}$ is a 1-dimensional vector representing the heuristic value for s, thus we enforce both V_{out} and E_{out} to be the empty set.

5.1 Architecture

A STRIPS-HGN is composed of three main HGN blocks: the encoding, processing (core), and decoding block. Our architecture follows a recurrent *encode-process-decode* design (Hamrick et al. 2018; Battaglia et al. 2018), as depicted in Figure 4. The input hypergraph G_{inp} is firstly encoded to

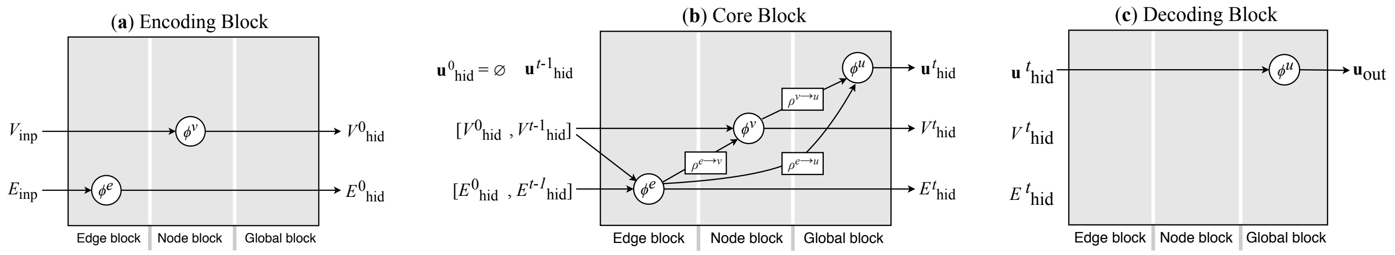

Figure 3: The encoding (a), core (b) and decoding (c) blocks of a STRIPS-HGN. The encoding block independently encodes the vertex and hyperedge features into latent space. The core block computes per-hyperedge and per-vertex updates using the concatenated input hypergraph, and additionally computes a latent heuristic feature $\mathbf{u}^t_{\text{hid}}$. The decoding block decodes the latent heuristic features $\mathbf{u}^t_{\text{hid}}$ into a single heuristic value.

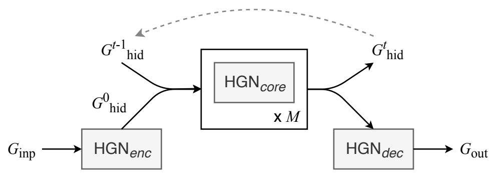

Figure 4: The recurrent encode-process-decode architecture of a STRIPS-HGN (modified from Figure 6c in Battaglia et al. 2018). The merging line for G^0_{hid} and G^{t-1}_{hid} indicates concatenation, while the splitting lines that are output by the HGN$_{core}$ block indicate copying (i.e., the same output is passed to different locations). The grey dotted line indicates that the output G^t_{hid} is used as input to the HGN$_{core}$ block in the next time step $t+1$.

a latent representation G^0_{hid} by the encoding block HGN$_{enc}$ at time step $t = 0$. This allows the network to operate on a richer representation of the input features in latent space.

Next, the initial latent representation of the hypergraph G^0_{hid} is concatenated with the previous output of the processing block HGN$_{core}$. Initially, when HGN$_{core}$ has not been called (i.e., at time step $t = 1$ just after G_{inp} has been computed), G^0_{hid} is concatenated with itself. Note that the hypergraph structure for G^0_{hid} and G^{t-1}_{hid} is identical because the HGN blocks do not update the senders or receivers for a hyperedge. Implementation-wise, concatenating a hypergraph with another involves concatenating the features for each corresponding vertex $\mathbf{v}_i$ together, and the features for each corresponding hyperedge $\mathbf{e}_k$ together (the global features are not concatenated as they are not required as input to a STRIPS-HGN). This results in a broadened feature vector for each vertex and hyperedge.

The processing block HGN$_{core}$, which outputs a hypergraph G^t_{hid} for each time step $t \in \{1, \ldots, M\}$, is applied M times with the initial encoded hypergraph G^0_{hid} concatenated with the previous output of HGN$_{core}$ as the input (see Figure 4). Evidently, this results in $M-1$ intermediate hypergraph outputs, one for each for time step $t \in \{1, \ldots, M-1\}$, and one final hypergraph for the time step $t = M$. The decoding block takes the hypergraph output by the HGN$_{core}$ block and decodes it to the hypergraph G_{out} which contains the heuristic value for state s in the global feature $\mathbf{u}_{\text{out}}$. Observe that we can decode each latent hypergraph which is output by HGN$_{core}$ to obtain a heuristic value for each time step $t \in \{1, \ldots, M\}$. We use this fact to train a STRIPS-HGN by optimising the loss on the output of each time step.

Core Block Details. We can interpret a STRIPS-HGN as a message passing model which performs M steps of message passing (Gilmer et al. 2017), as the shared processing block HGN$_{core}$ is repeated M times using a recurrent architecture. A single step of message passing is equivalent to sending a 'signal' from a vertex to its immediate neighbouring vertices. Although this means that a vertex only receives a 'signal' from other vertices at most M hops away, we theorise that this is sufficient to learn a powerful function which aggregates proposition and action features in the latent space.

In contrast to architectures such as ASNets and CNNs, which have a fixed receptive field that is determined by the number of hidden layers, the receptive field of a STRIPS-HGN is effectively determined by the number of message passing steps. Evidently, we can increase or decrease the receptive field of a STRIPS-HGN by scaling the number of message passing steps, hence providing a significant advantage over networks with fixed receptive fields.

Within-Block Design. The encoder block (Figure 3a) HGN$_{enc}$ encodes the vertex and hyperedge input features independently of each other using its ϕ^v and ϕ^e, respectively.

The core processing block of a STRIPS-HGN (Figure 3b) takes the concatenated vertex and hyperedge features from the latent hypergraphs G^0_{hid} and G^{t-1}_{hid} as input. ϕ^e computes per-hyperedge updates based on these hyperedge and vertex features. ϕ^v computes per-vertex updates based on the vertex features and the aggregated features of the hyperedges where the vertex is a receiver, which is computed using $\rho^{e\to v}$. Finally, ϕ^u uses the aggregated vertex and aggregated hyperedge features calculated with $\rho^{e\to v}$ and $\rho^{e\to u}$, respectively, to compute a latent representation for the heuristic value.

The decoder block (Figure 3c) takes the latent representation of the global features $\mathbf{u}^t_{\text{hid}}$ of the hypergraph returned by the core HGN block and uses its ϕ^u to decode it into a one-dimensional heuristic value. The vertex and hyperedge features are not used as $\mathbf{u}^t_{\text{hid}}$ already represents an aggregation of these features as computed by HGN$_{core}$.

The choice of learning model for the update functions ϕ^e, ϕ^v and ϕ^u within each block is not strict, as long as the model conforms to the input and output requirements. The choice of aggregation functions $\rho^{e\to v}$, $\rho^{e\to u}$, and $\rho^{v\to u}$ should be permutation invariant to the ordering of the inputs, otherwise different heuristic values could be obtained for different permutations of the same STRIPS problem. We detail our choice of update and aggregation functions in Section 6.1 which describes our experimental setup.

5.2 Training Algorithm

We consider learning a heuristic function h as a regression problem, where h ideally provides near-optimal estimates of the cost to go. We train our STRIPS-HGNs with the values generated by the optimal heuristic h^*. Given a set of training problems $\mathcal{P} = \{P_1, \dots, P_n\}$, we run an optimal planner for each $P_i \in \mathcal{P}$ to obtain optimal state-value pairs $(s, h^*(s))$. We then generate the delete-relaxed hypergraph G for P_i and the state s to get a training sample $(G, h^*(s))$. We denote by $\mathcal{T}$ the set containing all training samples.

We do not train our networks on the optimal heuristic for the relaxed plan h^+, as we believe that the STRIPS-HGNs may potentially learn a tighter value than h^+ (i.e., h^*).

Weight Optimisation. We use supervised learning and assume that each update function in the encoder, core, and decoder blocks of a STRIPS-HGN has some weights that need to be learned. For simplicity, we aggregate these weights into a single variable θ. Let h^θ be the heuristic learned by a STRIPS-HGN which is parameterised by the weights θ.

Recall that we can decode the latent hypergraph that is output by the core HGN block at each time step $t \in \{1, \dots, M\}$ into a heuristic value h_t^θ. Our loss function averages the losses of these intermediate outputs at each time step to encourage a STRIPS-HGN to find a good heuristic value in the smallest number of message passing steps possible (Battaglia et al. 2018). We use the mean squared error (MSE) loss function:

$$\mathcal{L}_\theta(\mathcal{B}) = \frac{1}{|\mathcal{B}|} \sum_{(G,\ h^*(s)) \in \mathcal{B}} \frac{1}{M} \sum_{t \in \{1, \dots, M\}} \left(h_t^\theta(G) - h^*(s)\right)^2$$

where $\mathcal{B} \subseteq \mathcal{T}$ is a minibatch within the entire training set $\mathcal{T}$, M is the number of message passing steps, and G is the input hypergraph for state s in a problem.

We use minibatch gradient descent (Li et al. 2014) to update the weights θ in the direction which minimises L_θ by using the gradient $d\mathcal{L}_\theta(\mathcal{B})/d\theta$. In a single epoch, we apply this update to every minibatch $\mathcal{B}$. We repeatedly apply more epochs until we reach a maximum number of epochs or exceed a fixed training time. During evaluation time, we use the heuristic value h_M^θ output at the last message step $t = M$.

5.3 Limitations of STRIPS-HGNs

Firstly, it is expensive to compute a single heuristic value using a STRIPS-HGN, given the computational cost of the matrix operations required for a single step of message passing; these costs scale with the number of vertices and hyperedges in the hypergraph. However, this cost may pay off if the learned heuristic provides very informative estimates near the optimal h^*, as it may reduce the total CPU time required to find a near-optimal solution.

The number of message passing steps M for the core HGN block is a hyperparameter which, in theory, should be adaptively selected based on how 'far' away the current state is from the goal. However, determining a good value for M is not trivial, and should ideally be automatically determined by a STRIPS-HGN by using its intermediate outputs. In practice, we found that setting $M = 10$ was sufficient to achieve promising results.

Finally, we are unable to provide any formal guarantees that the heuristics learned by STRIPS-HGNs are admissible. Although we train STRIPS-HGNs on the optimal heuristic values, it is unfeasible to analyse a network to understand what it is exactly computing.

6 Empirical Evaluation

Our experiments are aimed at showing the generalisation capability of STRIPS-HGNs to problems they were not trained on. For each experiment, we select a small pool of training problems (potentially from several domains) and train a STRIPS-HGN. We then evaluate the learned heuristic on a larger pool of testing problems with differing initial/goal states, problem sizes and even domains. We repeat each experiment for STRIPS-HGNs 10 times, resulting in 10 different trained networks, to measure the influence of the randomly generated problems and the training procedure.

6.1 Experimental Setup

Hardware. All experiments were conducted on an Amazon Web Services `c5.2xlarge` server with an Intel Xeon Platinum 8000 series processor running at 3.4Ghz. To ensure fairness between STRIPS-HGN and our baselines, each experiment was limited to a single core. We enforced a 16GB memory cutoff; however, only blind search reached this cutoff and the other planners never exceeded 2GB.

Search Configuration. We compare STRIPS-HGNs against the following baselines: no heuristic (i.e., blind), h^{max}, LM-cut, and h^{add}. These baselines all represent heuristics computable using the same input as used by STRIPS-HGNs— the delete-relaxation hypergraph — making this a fair comparison as all heuristics have access to the same information. We use A* as the search algorithm to compare the different heuristics, since STRIPS-HGNs are trained using optimal heuristic values and we believe that its estimates are sufficiently informative to find the optimal solution.

To generate the training data for each training problem, we used Fast Downward (Helmert 2006) configured with A* search and the LM-cut heuristic with a timeout of 2 minutes. To evaluate each testing problem with a heuristic, we used A* search in Pyperplan (Alkhazraji et al. 2011) with a 5 minute timeout. For each problem and heuristic, we run A* once.

We used Pyperplan for evaluation as STRIPS-HGNs are implemented in Python. We observed that the implementations of the delete-relaxation heuristics in Pyperplan are much slower than their counterparts in Fast Downward. Hence, our results for CPU times should be considered as preliminary.

STRIPS-HGNs Configuration. We generate the hypergraph of each planning problem by using the delete-relaxed problem computed by Pyperplan. For a STRIPS problem $P = \langle F, O, I, G, c\rangle$ and a given state $s \subseteq F$, we encode the input features for each proposition (vertex) $p \in F$ as a vector $[x_s, x_g]$ of length 2 where: $x_s = 1$ (resp. $x_g = 1$) iff p is true in state s (resp. in the goal G), and 0 otherwise. The input feature for each action $o \in O$ represented by a hyperedge e

is a vector $[w_e, r_e, s_e]$, where w_e is the cost $c(o)$ of o, and $r_e = |Add(o)|$ and $s_e = |Pre(o)|$ are the number of positive effects and preconditions for action o, respectively. r_e and s_e are used by a STRIPS-HGN to determine how much of a 'signal' it should send from a given hyperedge.

We set the number of message passing steps M for the recurrent core HGN block to 10, and implement each update function as a Multilayer Perceptron (MLP) with two sequential fully-connected (FC) layers, each with an output dimensionality of 32. We apply the LeakyReLU activation function (Maas, Hannun, and Ng 2013) following each FC layer. We add an extra FC layer with an output dimensionality of 1 in ϕ^u of the decoding block. Since the input to a MLP must be a fixed-size vector, we concatenate each update function's input features before feeding them into the MLP.

However, for the hyperedge update function ϕ^e in the core block, the number of receiver r_e and sender s_e vertices may vary with each hyperedge. For a given set of domains, we can compute the maximum number of preconditions N_{sender} and positive effects $N_{receiver}$ of each possible action by analysing their action schemas – this allows us to fix the size of the feature vectors for the receiver and sender vertices. We convert the set of input vertices $\mathbf{R}_k$ (resp. $\mathbf{S}_k$) into a fixed-size vector determined by $N_{receiver}$ (resp. N_{sender}), by stacking each vertex feature $\mathbf{v} \in \mathbf{R}_k$ (resp. $\mathbf{v} \in \mathbf{S}_k$) in alphabetical order by their proposition names, and padding the vector with zeros if the required length is not reached.

For the aggregation functions $\rho^{e \to v}$, $\rho^{e \to u}$ and $\rho^{v \to u}$ in the core block of a STRIPS-HGN, we use element-wise summation. We denote the heuristic learned by this configuration of STRIPS-HGN as h^{HGN}.

Training Procedure. We split the training data into n bins using quantile binning of the target heuristic values and use stratified k-fold to split the training set into folds $F = \{f_1, \dots, f_k\}$ with each fold containing approximately the same percentage of samples for each heuristic bin. For each fold $f \in F$, we train a STRIPS-HGN using $F \setminus f$ as the training set and f as the validation set, and select the network at the epoch which achieved the lowest loss on the validation set f. Since we train one STRIPS-HGN for each of the k folds, we are left with k separate networks. We select the network which performed best on its validation set as the single representative STRIPS-HGN for an experiment, which we then evaluate on a previously unseen test set.

Although k-fold is more commonly used for cross validation, we use it to reduce potential noise and demonstrate robustness over the training set used. Unless otherwise specified, we set $n = 4$ and $k = 10$ and use the Adam optimiser with a learning rate of 0.001 and a L2 penalty (weight decay) of 0.00025 (Kingma and Ba 2014). We set the minibatch size to 1 as we found that this resulted in a learned heuristic with the best planning performance and helped the loss function converge much faster despite the 'noisier' training procedure. This may be attributed to the small size of our training sets, which is usually limited to 50-200 samples.

6.2 Domains and Problems Considered

The actions in the domains we consider have a unit cost. The problems we train and evaluate on are randomly generated and unique. We consider the following domains:

- **8-puzzle** (Fern, Khardon, and Tadepalli 2011). Our training and test set consists of 10 and 50 problems, respectively. Only the initial state varies in these problems.
- **Sokoban** (Fern, Khardon, and Tadepalli 2011). Our training set consists of 20 problems (10 of grid sizes 5 and 7), and our test set contains 50 problems (20 of grid sizes 5 and 7, 10 of size 8). The number of boxes was set to 2 and the number of walls randomly selected between 3 and 5.
- **Ferry**[1]. Our training set consists of 9 problems, one for each of the parametrisations $\{2, 3, 4 \text{ locations}\} \times \{1, 2, 3 \text{ cars}\}$. Our test set contains 36 problems ($\{2, 3, \dots, 10 \text{ locations}\} \times \{5, 10, 15, 20 \text{ cars}\}$).
- **Blocksworld** (Slaney and Thiébaux 2001). Our training set for Blocksworld is formed of 10 problems (5 with 4 and 5 blocks, resp.). We have two evaluation sets for separate experiments: $\mathcal{P}_{bw1}$ consists of 100 problems (20 with $6, \dots, 10$ blocks, resp.), while $\mathcal{P}_{bw2}$ consists of 50 problems (10 with $4, \dots, 8$ blocks, resp.).
- **Gripper** (Long et al. 2000). Our training set for Gripper contains 3 Gripper problems (1, 2, 3 balls, resp.). Due to the low number of samples for Gripper (only 20 pairs), we resample the training set to 60 samples using stratified sampling with replacement. The test set consists of 17 problems with $4, \dots, 20$ balls.
- **Zenotravel** (Long and Fox 2003). Our training set consists of 10 problems (5 with 2 and 3 cities, resp., with 1-4 planes and 2-5 people), while our testing set contains 60 problems ($\{2, 3, 4 \text{ cities}\} \times \{2, 3, 4, 5 \text{ planes}\} \times \{3, 4, 5, 6, 7 \text{ passengers}\}$).

	Blind	h^{max}	h^{add}	LM-cut	h^{HGN}
8-puzzle	1	1	1	1	1
Sokoban	1	1	1	0.96	0.91
Ferry	0.42	0.36	1	0.47	0.77
Seen Blocksworld	0.78	0.68	1	0.97	0.97
Seen Gripper	0.71	0.59	0.59	0.41	0.69
Seen Zenotravel	0.62	0.55	1	0.82	0.6
Unseen Blocksworld	1	1	1	1	0.88

Table 1: Coverage Ratio (to 2 d.p.) on the test problems for each heuristic. The lower average coverage of $h^{HGN}$is attributed our noisy training procedure, which leads to trained networks with varying performance across experiments.

6.3 Experimental Results

Our experiments may be broken down into learning a domain-dependent or a domain-independent heuristic. For each of the experiments we describe below, we present the results for the number of nodes expanded, CPU time, and deviation from the optimal plan length when using A* (Figure 5). For h^{HGN}, the results are presented as the average and its 95% confidence interval over the 10 different experiments. Additionally, the coverage ratio on the testing problems for each heuristic is shown in Table 1. For h^{HGN}, we calculate the average coverage for the 10 repeated experiments.

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

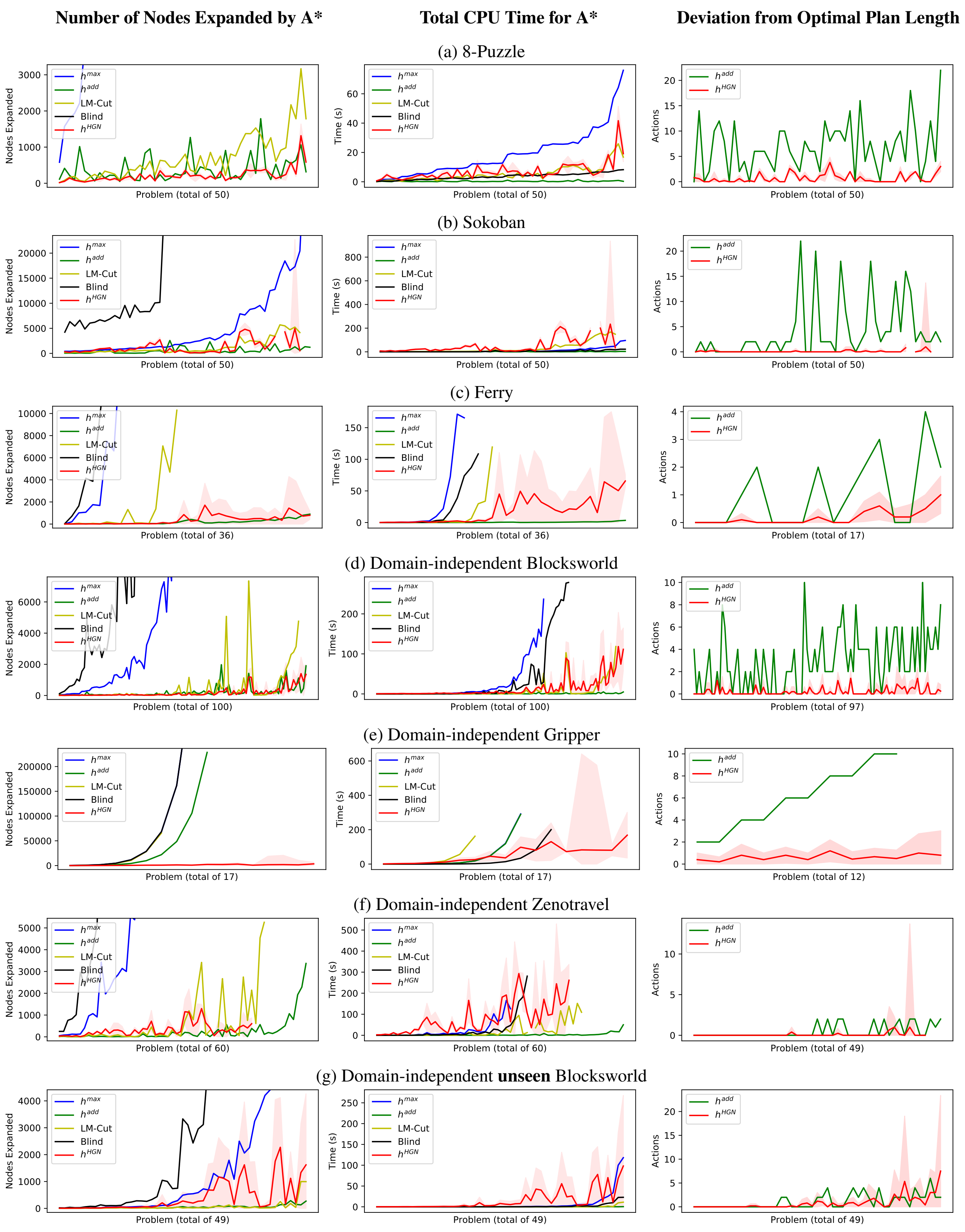

Figure 5: The plots for the results of our experiments described in Section 6.3. The discontinuities in the curves indicate problems which could not be solved within the search time limit. The pale red area indicates the 95% confidence interval for h^{HGN}, which may be very large if our experiments achieved low coverage. Problems for which no optimal solver was able to find a solution are omitted from deviation from optimal plan length plots.

Can we learn domain-dependent heuristics? In order to evaluate this, we train and test STRIPS-HGNs separately on 8-puzzle, Sokoban and Ferry. For 8-puzzle, we limit the training time for each fold to 10 minutes. For Sokoban, we use $n = 5$ bins and $k = 5$ folds and limit the training time within each fold to 20 minutes. For Ferry, we use $k = 5$ folds as the training set is quite small (61 samples) and limit the training time for each fold to 3 minutes.

Figures 5a to 5c depict the results of these experiments. Firstly, for 8-puzzle, h^{HGN} expands less nodes than all the baselines including h^{add}, yet h^{HGN} deviates significantly less from the optimal plan. For Sokoban, h^{HGN} expands marginally more nodes than h^{add} and LM-cut, but finds near-optimal plans. This is respectable, as Sokoban is known to be difficult for learning-based approaches as it is PSPACE-complete (Culberson C. 1997). Finally, for Ferry, h^{HGN} is able to solve problems of much larger size than the admissible heuristics are able to solve. h^{HGN} also obtains a smaller deviation from the optimal than h^{add}. Therefore, STRIPS-HGNs are able to learn domain-dependent heuristics which potentially outperform our baseline heuristics.

Can we learn domain-independent heuristics? To determine whether this is feasible, we train a STRIPS-HGN using data from multiple domains at once: the training set of each domain is binned and stratified into k-folds then, for $i \in \{1, \ldots, k\}$, the folds f_i of all considered domains are merged into a single fold $\hat{f}_i$ and $\hat{F} = \{\hat{f}_1, \ldots, \hat{f}_k\}$ is used as the training set. Using this procedure, we train a STRIPS-HGNs on the training problems for Blocksworld, Gripper and Zenotravel; and evaluate the network on the respective test sets for these domains ($\mathcal{P}_{bw1}$ for Blocksworld). We limit the training time for each fold to 15 minutes. Notice that each testing domain has been **seen** by the network during training.

Figures 5d to 5f depict our results. For Blocksworld, h^{HGN} requires fewer node expansions on average than all the baselines including h^{add}, which compared to h^{HGN}, deviates significantly more from the optimal plan length. For Gripper, h^{HGN} requires remarkably less node expansions than the baselines and is able to find solutions to the larger test problems within the limited search time (h^{max} and LM-cut are occluded by h^{add} for the number of nodes expanded). For Zenotravel, h^{HGN} requires fewer node expansions than the blind heuristic, h^{max} and LM-cut for the more difficult problems. However, at a certain point we are unable to solve more difficult problems due to the expense of computing a single heuristic value. Additionally, h^{HGN} deviates slightly less from the optimal plan length compared to h^{add}.

Thus, STRIPS-HGNs are capable of learning domain-independent heuristics which generalise to problems from the domains a network has seen during training. This is a very powerful result, as current approaches for learning domain-independent heuristics rely on features derived from existing heuristics, while we are able to learn heuristics from scratch.

Is h^{HGN} capable of generalising to unseen domains? To determine whether this is the case, we train each STRIPS-HGN on the training problems for Zenotravel and Gripper, while we evaluate the network on the test problems $\mathcal{P}_{bw2}$ for Blocksworld. We use the same training data generation procedure described for learning domain-independent heuristics, and limit the training time for each fold to 10 minutes.

Notice that Blocksworld is not in the training set, thus it is an **unseen** domain for h^{HGN}. Figure 5g depicts the results of this experiment (one problem for which h^{HGN} achieved low coverage is left out as it skews the plots). We can observe that h^{HGN} does better than h^{max} and blind search in terms of number of node expansions. This is despite the fact that the network did not see any Blocksworld problems during training. We note that we ran the experiments with unseen Gripper and unseen Zenotravel (using the other two domains as the training set). The results for these were not as promising compared to unseen Blocksworld, but the STRIPS-HGNs still managed to learn a meaningful heuristic: for Gripper, the STRIPS-HGNs perform similarly to the admissible heuristics, including no deviation from the optimal, but do not scale up to large problems; and for Zenotravel, STRIPS-HGN performs better than blind search and h^{max} but is outperformed by LM-cut and h^{add}.

This shows that it is possible for h^{HGN} to generalise across to problems from domains it has not seen during training. Unsurprisingly, h^{HGN} suffers a loss in planning performance in comparison to networks trained directly on the unseen domain.

Why is h^{HGN} not competitive in terms of CPU time? This may be attributed to our current sub-optimal implementation of STRIPS-HGNs, and the cost of evaluating the network (i.e., M message passing steps). Consequently, there is significant room for improvement in this regard. Despite this, our results show that STRIPS-HGNs is a feasible and effective approach for learning domain-dependent and domain-independent heuristics.

7 Conclusion and Future Work

We have introduced STRIPS-HGNs, a recurrent encode-process-decode architecture which uses the Hypergraph Networks framework to learn heuristics which are able to generalise not only across states, goals, and object sets, but also across unseen domains. In contrast to existing work for learning heuristics, STRIPS-HGNs are able to learn powerful heuristics from scratch, using only the hypergraph induced by the delete-relaxation of the STRIPS problem. This is achieved by leveraging Hypergraph Networks, which allow us to approximate optimal heuristic values by performing message passing on features in a rich latent space. Our experimental results show that STRIPS-HGNs are able to learn domain-dependent and domain-independent heuristics which are competitive with h^{max}, h^{add} and LM-cut, which are computed over the same hypergraph, in terms of the number of node expansions required by A*. This suggests that learning heuristics over hypergraphs is a promising approach, and hence should be investigated in further detail.

Potential avenues for future work include using a richer set of input features such as the *disjunctive action landmarks* or the *fact landmarks*. This may help the network learn a heuristic which is closer to the optimal and reduce the number of message passing steps required to obtain an informative heuristic estimate. Moreover, the time required to compute a single heuristic value ($\approx$0.01 to 0.02 seconds) could be reduced significantly by optimising our implementation (e.g., using multiple CPU cores or GPUs, optimising matrix operations and broadcasting), adapting the number of message passing steps in real time, or even pruning the vertices and hyperedges in the hypergraph of the relaxed problem.

Finally, we may investigate how to adapt STRIPS-HGNs for Stochastic Shortest Path problems (SSPs) (Bertsekas and Tsitsiklis 1991). Existing heuristics for SSPs either rely on linear programming (Trevizan, Thiébaux, and Haslum 2017), which can be expensive, or rely on *determinisation*, which oversimplifies the probabilistic actions. It may be possible to use Hypergraph Networks to learn an informative heuristic that preserves the probabilistic structure of actions by deriving suitable hypergraphs from *factored* SSPs.

References

Alkhazraji, Y.; Frorath, M.; Grützner, M.; Liebetraut, T.; Ortlieb, M.; Seipp, J.; Springenberg, T.; Stahl, P.; and Wülfing, J. 2011. Pyperplan.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2010. Bootstrap Learning of Heuristic Functions. In *Symposium on Combinatorial Search*.

Asai, M., and Fukunaga, A. 2018. Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary. In *AAAI*.

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

Bertsekas, D. P., and Tsitsiklis, J. N. 1991. An Analysis of Stochastic Shortest Path Problems. *Mathematics of Operations Research* 16(3):580–595.

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

Buffet, O., and Aberdeen, D. 2009. The factored policy-gradient planner. *Artificial Intelligence* 173(5-6):722–747.

Culberson C., J. 1997. Sokoban is PSPACE-complete. Technical report, University of Alberta.

Fern, A.; Khardon, R.; and Tadepalli, P. 2011. The first learning track of the international planning competition. *Machine Learning* 84(1-2):81–107.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

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

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural Message Passing for Quantum Chemistry. In *ICML*.

Gomoluch, P.; Alrajeh, D.; Russo, A.; and Bucchiarone, A. 2017. Towards learning domain-independent planning heuristics. In *IJCAI Workshop on Architectures for Generality and Autonomy*.

Groshev, E.; Tamar, A.; Goldstein, M.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies using Deep Neural Networks. In *AAAI Spring Symposium*.

Hamrick, J. B.; Allen, K. R.; Bapst, V.; Zhu, T.; McKee, K. R.; Tenenbaum, J. B.; and Battaglia, P. W. 2018. Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*.

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

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *ICAPS*.

Jiménez Celorrio, S.; de la Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A Review of Machine Learning for Automated Planning. *Knowledge Eng. Review* 27(4):433–467.

Kingma, D. P., and Ba, J. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.

Li, M.; Zhang, T.; Chen, Y.; and Smola, A. J. 2014. Efficient Mini-batch Training for Stochastic Optimization. In *KDD*.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59.

Long, D.; Kautz, H.; Selman, B.; Bonet, B.; Geffner, H.; Koehler, J.; Brenner, M.; Hoffmann, J.; Rittinger, F.; Anderson, C. R.; et al. 2000. The AIPS-98 planning competition. *AI Magazine* 21(2):13.

Maas, A. L.; Hannun, A. Y.; and Ng, A. Y. 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *ICML*.

Richter, S., and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Samadi, M.; Felner, A.; and Schaeffer, J. 2008. Learning from Multiple Heuristics. In *AAAI*.

Say, B.; Wu, G.; Zhou, Y. Q.; and Sanner, S. 2017. Nonlinear Hybrid Planning with Deep Net Learned Transition Models and Mixed-Integer Linear Programming. In *IJCAI*.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019. Deep Learning for Cost-Optimal Planning: Task-Dependent Planner Selection. In *AAAI*.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125(1-2):119–153.

Thayer, J. T.; Dionne, A. J.; and Ruml, W. 2011. Learning Inadmissible Heuristics During Search. In *ICAPS*.

Toyer, S.; Trevizan, F. W.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies With Deep Learning. In *AAAI*.

Toyer, S.; Trevizan, F. W.; Thiébaux, S.; and Xie, L. 2019. ASNets: Deep Learning for Generalised Planning. *arXiv preprint arXiv:1908.01362*.

Trevizan, F.; Thiébaux, S.; and Haslum, P. 2017. Occupation measure heuristics for probabilistic planning. In *ICAPS*.