Learning Heuristics for Planning with Hypergraph Networks

William Shen

A thesis submitted for the degree of
Bachelor of Advanced Computing
(Research and Development, Honours)
The Australian National University

Except where otherwise indicated, this thesis is my own original work.

William Shen

To my parents.

Acknowledgments

Firstly, I would like to express my sincere gratitude towards my supervisors Felipe Trevizan and Sylvie Thiébaux for their continuous support, guidance and patience throughout my honours project. Their indispensable feedback and advice has helped me appreciate and navigate the world of deep learning and planning, and consider whether research is the right career path.

I am also grateful to the Australian National University for funding my studies for the duration of my degree through the National University Scholarship, and to the College of Engineering and Computer Science and my supervisors for providing me with the opportunity to attend and present my work at AI conferences.

Finally, I wish to thank my family who have continuously supported my education and encouraged me to pursue my passions and interests.

Abstract

Planning is the fundamental ability of an intelligent agent to reason about what decisions it should make in a given environment to achieve a certain set of goals. State-of-the-art planners use forward-chaining state space search guided by a heuristic. This method of search incrementally builds the search tree from the initial state until a goal is reached, whilst the heuristic is used to guide the search algorithm to what the heuristic identifies as promising parts of the search space.

Deep Learning harnesses the power of neural networks to automatically learn powerful features and knowledge from experience. Although deep learning has gained immense popularity in fields including computer vision and natural language processing, there is still no consensus as to what deep learning architecture is best for reasoning and decision making tasks such as planning. Recent work in deep learning for planning is primarily concerned with learning which planner to apply for a given problem (planner selection), or learning generalised policies which are functions that select which action should be applied by an agent in a given state.

In contrast, this thesis focuses on how we may harness the power of deep learning to learn heuristic functions which exploit the structure of planning problems. We investigate how we may learn heuristics which generalise to problems of larger size than the problems a neural network was trained on within a single domain (i.e., a domain-dependent heuristic). Additionally, we explore the feasibility of learning domain-independent heuristics which are able to generalise across domains.

Our work makes three key contributions. Our first contribution is Hypergraph Networks (HGNs), our novel framework which generalises Graph Networks [Battaglia et al., 2018] to hypergraphs. A hypergraph is a generalisation of a normal graph in which a hyperedge may connect any number of vertices together. 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. Our second contribution is STRIPS-HGNs, an instance of a Hypergraph Network 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 allow them to incrementally propagate messages within the hypergraph in latent space. Our third and final contribution is our 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.

Contents

Chapter 1

Introduction

Planning is the fundamental ability of an intelligent agent to reason about what decisions it should make in a given environment to achieve a certain set of goals [Geffner and Bonet, 2013]. Deep learning is a sub-field within Artificial Intelligence (AI) where deep neural networks are trained to learn incredibly rich and complex functions from experience.

This thesis explores how deep learning may be used to learn both *domain-dependent* and *domain-independent* heuristics for planning.

1.1 Planning

We are concerned with classical planning, where the environment is fully-observable by the agent, and actions are deterministic [Geffner and Bonet, 2013]. More specifically, our work considers planning problems represented as STRIPS instances [Fikes and Nilsson, 1971].

In a planning problem, an agent must reason about which actions it should take from the initial state in order to achieve a certain set of goals. In doing so, the agent must consider how certain actions may affect its future performance. The solution to a planning problem is a sequence of actions which move the agent from the initial state into a goal state. We call this solution a plan $\pi = a_0, \ldots, a_n$ where each a_i represents the action applied at time step i.

State-of-the-art classical planning algorithms consider planning as forward-chaining state space search guided by a *heuristic*. A forward-chaining state space search algorithm incrementally builds a search tree from the initial state until it reaches a goal state. A heuristic is a function which provides estimates of the cost to reach a goal state from a given state. An informative heuristic usually helps a search algorithm find a plan in a smaller number of node expansions, as it guides the search to more promising parts of the search space.

Planning has proven to be an immensely important field within symbolic AI with a variety of real-world applications. Classical planning has been used to control industrial printers [Ruml et al., 2011], analyse network vulnerabilities [Boddy et al., 2005], and even plan activities for rovers on Mars [Bresina et al., 2005].

1.2 Deep Learning

The deep learning (DL) 'revolution' refers to the unprecedented rate at which DL models have become the de facto state-of-the-art algorithms in several fields within the past decade. For example, Convolutional Neural Networks (CNNs) have become the standard in computer vision, as they are able to automatically learn filters to apply to local neighbourhoods across an entire image [Krizhevsky et al., 2012; LeCun et al., 1998]. Recurrent Neural Networks, including Long Short-Term Memory [Hochreiter and Schmidhuber, 1997], have become extremely powerful for Natural Language Processing (NLP) as they are able to learn the dynamics of sequential input [Young et al., 2018].

The majority of existing deep learning architectures are designed for tasks which are inherently *perception*-based. Loosely speaking, perception refers to the ability of a neural network to interpret and understand data in a similar way to humans, and generate a corresponding transformation. Examples of perception tasks could include labelling objects in an image [Krizhevsky et al., 2012], or automatically classifying the sentiment of a sentence [Joulin et al., 2016].

1.3 Deep Learning for Planning

Despite the prevalence of DL models for perception-based problems, there is still no consensus as to what DL architecture is best for reasoning and decision making tasks such as planning. The proposed deep learning approaches for planning can be split into three categories: planner selection, learning generalised policies, and learning heuristics.

An example of deep learning applied to planner selection is Sievers et al. [2019], who train Convolutional Neural Networks (CNNs) over graphical representations of planning problems to determine which planner should be invoked for a planning task. For learning generalised policies and heuristics, Groshev et al. [2018] use CNNs and Graph Convolutional Networks and show that they are able to generalise to problems they were not trained on. Notice that both of these approaches use standard deep learning architectures applied for planning.

In contrast, Action Schema Networks [Toyer et al., 2019] define a dedicated neural network architecture which exploits the relational structure of planning problems encoded in (P)PDDL [Younes and Littman, 2004] to learn generalised policies for classical and probabilistic planning problems.

The motivation of this thesis is to use and extend existing deep learning architectures to planning. Unlike existing approaches to learning for planning which rely on hand-engineering features or encoding planning problems as images, we present a domain-independent learning algorithm which automatically extracts knowledge and features from a STRIPS problem. This knowledge is represented as a *hypergrpah*.

1.4 Contributions and Research Goals

The main objective of this thesis is to investigate how we may use deep learning to learn heuristics by exploiting the structure of a planning problem. Our primary goal is to propose a domain independent algorithm that can be used for learning both domain-dependent and domain-independent heuristics. A domain-dependent heuristic is able to generalise across problems in a given domain, while a domain-independent heuristic is able to generalise across problems in multiple domains.

We learn heuristics instead of learning policies, i.e., actions to apply in a given state, as a heuristic may be combined with a search algorithm that provides formal guarantees. For example, A* search is *complete* meaning that it guarantees a plan will be eventually found if one exists, regardless of the heuristic function used. This means that learned heuristics can be used in critical applications, as a heuristic search algorithm can easily correct for any deficiencies or misleading information. Although it is possible to combine a search algorithm with a neural network which learns a probability distribution over actions [Shen et al., 2019], learning heuristics provides a much more efficient and lighter layer of reasoning . This thesis makes three key technical contributions:

1. **Hypergraph Networks**
   We introduce Hypergraph Networks (HGNs), our generalisation of Graph Networks [Battaglia et al., 2018] to hypergraphs. A hypergraph is a generalisation of a normal graph in which a hyperedge may connect any number of vertices together. HGNs may be used to design new deep learning models which operate over hypergraphs, and inherently support combinatorial optimisation by applying per-hyperedge and per-vertex updates. Moreover, HGNs are designed to be highly flexible, and can be used to represent existing hypergraph deep learning models.

2. **STRIPS-HGNs: a Hypergraph Network for Learning Heuristics**
   STRIPS-HGNs is an instance of a Hypergraph Network which is designed to learn heuristics by approximating shortest paths over the hypergraph induced by the *delete relaxation* of a STRIPS problem. A STRIPS-HGN uses a recurrent *encode-process-decode* architecture to incrementally propagate the *latent* vertex and hyperedge features by using *message passing*.

3. **Extensive Empirical Evaluation**
   We train and evaluate our STRIPS-HGNs on a variety of domains. Our experiments show that STRIPS-HGNs are able to learn knowledge from the features and structure in a hypergraph which helps it generalise to problems much larger than the problems a network was trained on. We also show that it is possible for a STRIPS-HGN to generalise to problems from a domain it was not trained on. In contrast to the majority of existing techniques for learning heuristics in planning, STRIPS-HGNs learn heuristics from scratch and do not use input features computed from domain-independent heuristics.

1.5 Thesis Outline

The structure of the remainder of this thesis is as follows:

- **Chapter 2 – Background and Related Work**. The main objective of this chapter is to provide the reader with the relevant background necessary to understand both planning and deep learning. Firstly, we formalise classical planning, discuss heuristic search, and investigate how existing delete-relaxation heuristics are computed. We then present the *Multilayer perceptron* and describe how it may be trained. This is followed by a review of existing neural network models which operate over standard graphs. We conclude Chapter 2 with an reasoned study of existing work for machine learning and deep learning applied to planning.

- **Chapter 3 – Hypergraph Networks**. In this chapter, we formally define what a hypergraph is and then discuss existing deep learning models which operate over hypergraphs. The main contribution in Chapter 3 is to introduce Hypergraph Networks (HGNs), our novel framework which generalises Graph Networks [Battaglia et al., 2018] to hypergraphs.

- **Chapter 4 – Learning Heuristics over Hypergraphs**. We firstly discuss how h^{max} and h^{add} may be considered as shortest path problems over hypergraphs. The main objective of Chapter 4 is to introduce STRIPS-HGNs, our specific instance of a Hypergraph Network which emulates *message passing* using a recurrent *encode-process-decode* architecture. We discuss the inherent combinatorial generalisation capabilities built into a STRIPS-HGN, and present an algorithm for generating optimal training data and optimising the weights of the network.

- **Chapter 5 – Empirical Evaluation**. Chapter 5 firstly describes our experimental setup in detail, including particulars regarding the configurations of our Hypergraph Networks and our training procedure which aims to reduce noise. Next, we introduce the domains we evaluate our STRIPS-HGNs on and the classes of heuristics which we aim to learn. The remainder of Chapter 5 analyses and explains the results of our experiments, which show that STRIPS-HGNs are able to learn domain-dependent and domain-independent heuristics.

- **Chapter 6 – Conclusion**. We conclude this thesis by summarising our contributions, and discussing several promising directions for future work.

Chapter 2

Background and Related Work

This chapter aims to provide the reader with the relevant background necessary to understand planning and deep learning. Section 2.1 firstly formalises classical planning, then discusses heuristic search, and closes by presenting domain-independent heuristics for planning.

Section 2.2 then provides a brief introduction to neural networks, how they are trained, and a study of graph neural networks. Section 2.3 concludes this chapter by discussing current machine learning and deep learning approaches applied to planning.

2.1 Planning

Planning is the fundamental ability of an intelligent agent to reason about what decisions it should make in a given environment to achieve a certain set of goals [Geffner and Bonet, 2013]. In less abstract terms, the agent's objective is to apply a sequence of actions that moves it from the initial state to a goal state, ideally with minimal cost.

In this thesis, we are mainly concerned with classical planning, where the environment is fully-observable by the agent, and applying an action can only lead to one outcome (i.e., actions are deterministic). This is in contrast to probabilistic planning, where actions are stochastic and may lead to one of several outcomes which is sampled according to a predefined state transition distributed. [Mausam and Kolobov, 2012].

2.1.1 Representations in Planning

Classical Planning

A classical planning problem may be described as the state model $\mathcal{S} = \langle S, s_0, S_G, A, f, c \rangle$ [Geffner and Bonet, 2013], where:

- S is the finite set of states;
- $s_0 \in S$ is the initial state;

- $S_G \subset S$ is the non-empty set of goal states;
- A is the finite set of actions and $A(s) \subseteq A$ indicates the actions which are applicable in a given state $s \in S$;
- $f(a, s)$ is the transition function which indicates the new state $s' = f(a, s)$ after applying action $a \in A(s)$ in state $s \in S$; and
- $c(a, s)$ is cost of applying action $a \in A(s)$ in the state s.

A solution to a classical planning problem is a plan $\pi = a_0, \ldots, a_n$ (i.e., a sequence of actions) which generates a state sequence $s_0, s_1, \ldots, s_{n+1}$ where $s_{n+1} \in S_G$ is a goal state [Geffner and Bonet, 2013].

The cost of a plan is the sum of the costs of applying each action in the plan $\sum_{i \in \{0,\ldots,n\}} c(a_i, s_i)$. An optimal plan is a plan which has minimum cost. This thesis considers classical planning problems where actions have a unit cost of $c(a, s) = 1$ for all $s \in \mathcal{S}$ and $a \in A(s)$. Subsequently, the cost of a plan is now given by its length.

STRIPS Representation

STRIPS is a language for representing classical planning problems which have boolean state variables [Fikes and Nilsson, 1971]. Thus, a proposition in STRIPS is a fact which must either be true or false. We define a STRIPS problem as 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 is a function $c(o)$ mapping an action $o \in O$ to a cost.

Each action $o \in O$ is defined as a triple $\langle Pre(o), Add(o), Del(o) \rangle$ where $Pre(o)$ represents the set of propositions which must be true in order for o to be applied, while $Add(o)$ and $Del(o)$ represent the sets of propositions which action o make true and false after applying o, respectively. We call $Pre(o)$ the preconditions, $Add(o)$ the additive effects, and $Del(o)$ the delete effects. It may be easily observed that STRIPS encodes the state model $\mathcal{S} = \langle S, s_0, S_G, A, f, c \rangle$ of a classical planning problem.

We define the delete relaxation of a STRIPS problem P as the STRIPS problem $P^+ = \langle F, O', I, G, c \rangle$, where $O' = \{\langle Pre(o), Add(o), \emptyset \rangle \mid o \in O\}$. Evidently, delete relaxation ignores the negative effects of each action in P.

It is important to note that classical planning is very difficult in terms of computational complexity. Bylander [1994] showed that determining whether a STRIPS problem has a plan is PSPACE-complete. In contrast, deciding whether a plan exists for the delete relaxation is a polynomial time problem, and determining whether a plan of length less than a given bound exists is NP-complete.

Planning Domain Description Language (PDDL)

The Planning Domain Definition Language (PDDL) is a standardised language which may be used to represent STRIPS problems [Ghallab et al., 1998]. Problems specified in PDDL consist of two parts: the *domain* definition and a *problem* definition.

The domain definition contains the action schemas and predicates which define a 'template' for instantiating a specific problem.

For example, car-at($?loc$) is a predicate which describes whether a car is at the location variable $?loc$. Now, drive($?loc1, ?loc2$) is an action schema for driving between the location variables $?loc1$ and $?loc2$, where both $?loc1$ and $?loc2$ are free variables. This action schema requires car-at($?loc1$) to be true (precondition), and generates a new state in which car-at($?loc1$) is false and car-at($?loc2$) is true.

A problem definition specifies the object names which replace the variables in the domain definition, along with the description of the propositions in the initial state and in the set of goal states. Grounding is then used to convert the domain definition and problem definition into a STRIPS problem.

2.1.2 Planning as Heuristic Search

Having discussed the representation of planning problems, we now investigate heuristic search, a strategy used by the majority of state-of-the-art-planners. Classical planning may be viewed as a path finding problem over a directed graph. The vertices (nodes) represent the states in a problem, while the edges represent actions. Search algorithms for this view of classical planning are known as state-space planners. Since the state space is exponential in the number of propositions in a planning problem and cannot be generated upfront and entirely explored, state space planners exploit the STRIPS representation of the problem to incrementally generate this directed graph as more search is performed [Geffner and Bonet, 2013].

Heuristic search planners are a special case of state-space planners that rely on forward-chaining search from the initial state to a goal state using a heuristic h, typically computed from the STRIPS representation, and which provides an estimate of the cost to reach a goal from a given state to guide the search to promising regions of the state space [Bonet and Geffner, 2001]. Many state-of-the-art planners are based on heuristic search algorithms – we review two heuristic search algorithms below.

A* Search

A* is a best-first search algorithm which at each search step, expands the node n which maximises the heuristic value $h(n)$ plus the cost of the path $g(n)$ from the root node for the initial state to n [Hart et al., 1968]. That is, A* selects the node that minimises $f(n) = g(n) + h(n)$ for expansion. The intuition behind A* is that it avoids expanding paths that are already known to be expensive [Russell and Norvig, 2016].

A* is guaranteed to terminate and is *complete*, that is, it always finds a plan given one exists, no matter how uninformative the heuristic is. It can also be proven that if the heuristic function h is *admissible*, then A* is guaranteed to return the optimal plan

[Hart et al., 1968]. An admissible heuristic is a heuristic which never overestimates the true optimal cost from any state to a goal. We will formalise heuristics in Section 2.1.3.

Greedy Best-First Search (GBFS)

GBFS selects the node that minimises $f(n) = h(n)$ for expansion In GBFS, nodes are selected for expansion based solely on their heuristic value – i.e., the cost to reach a node n ($g(n)$ in A*) is ignored when choosing a node to expand.

GBFS is a satisficing search algorithm which usually finds a plan much faster than A*, given its greedy behaviour. However, these plans are not guaranteed to be optimal. Because of this, we focus on using A* as the search algorithm in our experiments.

2.1.3 Heuristics

Heuristic functions allow a heuristic search algorithm to focus its search on what the heuristic believes to be promising parts of 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 given state s. The optimal heuristic $h^*(s)$ is the heuristic that gives the optimal cost to reach a goal state from s. A heuristic h is said to be admissible if it never overestimates the cost of reaching a goal, i.e., $\forall s \in \mathcal{S}\ h(s) \leq h^*(s)$. A heuristic that is not admissible is called inadmissible.

As we discussed in Section 2.1.2, A* is only guaranteed to find the optimal plan if used with an admissible heuristic. Despite this, a search algorithm often finds a solution much faster when used with an non-admissible rather than an admissible heuristic. This may be attributed to the fact that inadmissible heuristics can reduce the expanded search space by overestimating the cost of states it deems non-promising, i.e. by setting $h(s) \gg h^*(s)$ it can avoid expanding s.

This thesis considers three baseline domain-independent heuristics which are based on delete-relaxation: h^{max} (admissible), h^{add} (inadmissible) [Haslum and Geffner, 2000], and the Landmark-Cut heuristic (admissible) [Helmert and Domshlak, 2009].

Delete-Relaxation Heuristics

Delete-relaxation approximate the optimal heuristic value in a relaxed version of a planning problem where the negative effects of each action are ignored. This means that once a proposition becomes true, it will always stay true. Recall from Section 2.1.1 that the delete relaxation for a STRIPS problem $P = \langle F, O, I, G, c\rangle$ is the STRIPS problem $P^+ = \langle F, O', I, G, c\rangle$, where $O' = \{\langle Pre(o), Add(o), \emptyset\rangle \mid o \in O\}$.

Since computing the optimal heuristic value for a state s in P^+ is an NP-complete problem, h^{max} approximates this value as the cost of achieving the most expensive proposition in G starting from s. On the other hand, h^{add} approximates the optimal heuristic value for a state s in P^+ as the sum of the costs of achieving each proposition in G independently of the others. h^{max} is admissible while h^{add} is inadmissible. We

detail the computation of h^{max} and h^{add} in Section 4.1.1, where we also discuss how they may be viewed as approximating shortest paths over hypergraphs.

The LM-cut heuristic [Helmert and Domshlak, 2009] starts from the observation that the optimal cost of solving P is the minimum-cost of any *hitting set* for a complete set of *disjunctive action landmarks* for P^+. A disjunctive action landmark for a planning problem is a set of actions, at least one of which must be present in any plan solving the problem. Therefore any plan must "hit" all the disjunctive action landmarks for the problem. The optimal plan corresponds to selecting the cheapest set of actions that hits them all. Since computing a minimum cost-hitting set is NP-complete and the number of disjunctive action landmarks can be exponential in the problem size, LM-cut approximates the solution to this problem in a way that guarantees admissibility. It does so via a series of h^{max} computations for increasingly relaxed problems starting from P^+, and by incrementally finding a useful and sufficient set of landmarks that correspond to cuts in a *justification graph*. Although LM-cut may be more expensive to compute than h^{max}, the heuristic estimates it provides are generally much more informative.

Other Families of Heuristics

Although this thesis focuses on delete-relaxation heuristics, it is important to note that there are other families of heuristics which are widely used in planning.

Pattern Database Heuristics (PDBs) use lookup tables to store optimal heuristic values for an *abstraction* (pattern) of a problem [Haslum et al., 2007; Geffner and Bonet, 2013]. This abstraction is obtained by projecting the state space of a problem down to a subset of variables. PDBs then precompute the optimal values for these abstractions, and use these precomputed values for different patterns to guide the search towards states matching patterns with small costs.

Potential heuristics compute weighted linear combinations over a set of features for a state [Pommerening et al., 2015]. These features may be automatically learned from a small set of training instances by using mixed integer linear programming to give a domain-dependent heuristic [Francès et al., 2019].

2.2 Deep Learning

Deep learning (DL) has proven to be immensely successful in several fields including computer vision and natural language processing, where DL models have become the de facto state-of-the-art algorithm. This immense success may be attributed to several factors including the increased power of Graphical Processing Units (GPUs) and the wide-spread availability of big data.

This section aims to provide the reader with the necessary background for understanding the deep learning by firstly introducing multilayer perceptrons and discussing how they may be trained with *backpropagation* and *gradient descent*. We then discuss the concept of *relational inductive biases*, and conclude this section with a study of existing neural networks which operate over graphs.

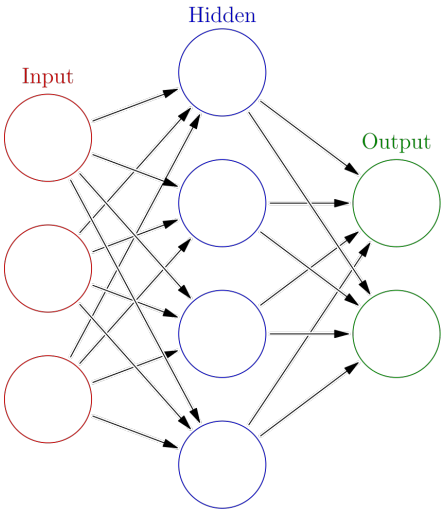


Figure 2.1: An example of a multilayer perceptron with three neurons in the input layer, a single hidden layer with 4 neurons, and two neurons in the output layer. Each arrow represents a weight in the MLP which needs to be learned. Diagram from https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg

2.2.1 Multilayer Perceptrons

Multilayer perceptrons (MLPs) are the simplest form of *feedforward* neural networks which contain fully connected layers of *neurons*. An example of a MLP with one *hidden* layer is depicted in Figure 2.1. A feedforward neural network is a network in which information only moves forward in one direction. This is in contrast to a *recurrent* neural network, which has lateral or feedback connections. A neuron represents a function which performs a weighted summation on its input and then applies an *activation function*.

The first layer in a MLP receives an input feature vector which is then fed through one or more intermediate *hidden* layers to compute hidden representations for each layer. The output of the final hidden layer is then fed through the output layer to get the output of the MLP. A hidden layer performs weighted summations of the hidden representations from the previous layer to compute a new hidden representation. This hidden representation is also called the **latent** representation, as it encodes features which are not easily interpretable or understood.

Throughout this section, we follow the notation defined in Toyer [2017]. In formal terms, the hidden representation $h^{(l)}$ which is output by the l-th layer in a MLP is computed as:

$$h^{(l)} = f(W^{(l)}h^{(l-1)} + b^{(l)})$$

where $h^{(l-1)}$ is the hidden representation of the previous layer (we define $h^{(0)}$ to be the input features), $W^{(l)}$ are the weights and $b^{(l)}$ are the biases of the neurons in the l-th layer. We may aggregate the weights $W^{(l)}$ and biases $b^{(l)}$ across the L layers $l \in \{1, \ldots, L\}$ in a MLP into a single variable θ, which we call the **weights** of a MLP.

f is an activation function which applies a non-linearity to the weighted summation, such as the sigmoid function or the Rectified Linear Unit (ReLU) [Nair and Hinton, 2010].

As we increase the number of hidden layers in a MLP, the neural network is progressively able to extract more complex features from the input. However, a larger network leads to an increased number of weights and biases which we would need to *learn*, and would increase the computation cost of training and evaluating a MLP.

2.2.2 Training a Neural Network

This thesis is only concerned with supervised learning, where a neural network learns from a labelled training dataset $\mathcal{T} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where each (x_i, y_i) represents the (input features, desired outputs). The objective of a MLP is to optimise its weights θ to minimise the total *loss* $\mathcal{L}(\mathcal{T})$ (Equation 2.1). The loss function ℓ measures the deviation of the predictions provided by a neural network when we feed it with the input features $\{x_1, \ldots, x_n\}$, to the targets features $\{y_1, \ldots, y_n\}$. We define the total loss as

$$\mathcal{L}(\mathcal{T}) = \sum_{x_i, y_i \in \mathcal{T}} \ell(\hat{y}_i, y_i) \tag{2.1}$$

where $\hat{y}_i = MLP(x_i)$ is the prediction made by the MLP for the input feature x_i. In order to train a neural network using *backpropagation* and *gradient descent*, the loss function ℓ must be differentiable. This thesis only considers the mean squared error (MSE) loss function, which is defined as $\ell_{MSE} = |\hat{y}_i - y_i|^2$.

Backpropagation and Gradient Descent

The backpropagation algorithm exploits the chain rule to update the weights of a neural network in a backwards manner, starting from the output layer and ending at the input layer of a network. We refer the reader to [Goodfellow et al., 2016] for a detailed discussion on backpropagation.

Given a differentiable loss function ℓ, we can compute the gradient $\frac{d\mathcal{L}_\theta(\mathcal{T})}{d\theta}$ of the total loss with respect to the weights θ of the network. The weights θ are then updated in the direction which minimises $\mathcal{L}(\mathcal{T})$ according to this gradient. This weight update approach is called batch gradient descent. We discuss other forms of gradient descent in Section 4.3.

We repeat this weight update procedure for several *epochs* until the neural network reaches a satisfactory total loss $\mathcal{L}(\mathcal{T})$. An epoch refers to a single pass over all the samples in the training dataset $\mathcal{T}$.

2.2.3 Relational Inductive Biases

Inductive biases refer to the assumptions that a learning algorithm makes which allow it to generalise beyond the finite set of data it was trained on.

Component	Entities	Relations	Rel. inductive bias	Invariance
MLPs	Neurons	All-to-all	Weak	-
Convolutional	Grid elements	Local	Locality	Spatial translation
Recurrent	Timesteps	Sequential	Sequentiality	Time translation
Graph network	Vertices	Edges	Arbitrary	Vertex, edge permutations

Table 2.1: Relational inductive biases in standard deep learning components. Modified from Table 1 in [Battaglia et al., 2018].

Relational inductive biases extend this notion and refer to the inductive biases which impose constraints on the relationships and interactions of the entities in a learning algorithm [Battaglia et al., 2018]. A learning algorithm has a strong relational inductive bias if it imposes little to no constraints on these relationships and interactions. Table 2.1 summarises the relational inductive biases in standard deep learning components; we refer the reader to [Battaglia et al., 2018] for a more detailed discussion.

The relational inductive biases of MLPs are very weak, as they do not impose any constraints on the interactions between neurons in layer l and neurons in layer $l+1$ as they are fully connected. This means that all neurons in a fully connected layer may be used to determine the output of the layer. On the other hand, Convolutional Neural Networks (CNNs) have strong relational inductive biases as they impose *locality* and *translation invariance* by applying the same filters in local neighbourhoods within grid data (e.g., images) [Krizhevsky et al., 2012; LeCun et al., 1998].

Graph Networks (GNs) [Battaglia et al., 2018], which we briefly describe in Section 2.2.4, impose much stronger relational inductive biases than CNNs. This is because GNs operate over graphs which represent arbitrary relational structures, while CNNs are only able to operate over grid-based structures.

2.2.4 Deep Learning on Graphs

There is an increasing interest in applications of deep learning to non-Euclidean domains and problems which may inherently be formalised as a graph. However, the complexity of graph-based data means that it is difficult for learning algorithms to capture the complex interdependence of entities within a graph. This subsection provides a brief study of current state-of-the-art *spectral* and *spatial*-based graph neural networks (GNNs). For more insights into GNNs, we refer the reader to the following survey [Wu et al., 2019].

Spectral-based Approaches

Spectral-based graph neural networks define convolutions based on spectral graph theory. We will not derive the formulas we use here, as the derivations require extensive knowledge on graph signal processing. We refer the reader to [Wu et al., 2019] for the formal derivations.

The graph Laplacian is a mathematical representation of an undirected graph which has several applications in spectral graph theory, including the ability to construct low dimensional embeddings for vertices. The graph convolution of an input graph signal $\mathbf{x}$ with a filter $\mathbf{g} \in \mathbb{R}^N$ is defined as $\mathbf{x} *_G \mathbf{g}_\theta$ [Wu et al., 2019], where $*_G$ represents the convolution on the graph G. The objective of a spectral GNN is to define a filter $\mathbf{g}_\theta$.

Bruna et al. [2013] define $\mathbf{g}_\theta$ as a set of learnable parameters with multiple channels which uses the eigen decomposition of the normalised graph Laplacian. Their approach cannot be applied to graphs with different structures as the eigenvectors and eigenvalues vary significantly even with small perturbations to a graph. Moreover, computing the eigen decomposition is cubic in the number of vertices n in the graph, i.e., $O(n^3)$, meaning that their approach is unscalable to larger graphs.

Defferrard et al. [2016] approximate $\mathbf{g}_\theta$ by using *Chebyshev polynomials* and the graph Laplacian. Their approach is linear in the number of edges m, i.e., $O(m)$, and provides a significant improvement over the $O(n^3)$ computational complexity of [Bruna et al., 2013].

Spectral-based Graph Convolutional Networks (GCNs) [Kipf and Welling, 2017] further approximate $\mathbf{g}_\theta$ with the first-order approximation of *Chebyshev polynomials* and a renormalisation trick. This results in the following layer-wise propagation rule for the l-th GCN layer:

$$\mathbf{X}^{(l+1)} = \sigma\left(\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}\mathbf{X}^{(l)}\mathbf{W}^{(l)}\right)$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ is the adjacency matrix of the undirected graph with self-connections, $\mathbf{I}_N$ is the identity matrix of size N, $\tilde{\mathbf{D}}[i,i] = \sum_j \tilde{\mathbf{A}}[i,j]$ is the square degree containing the degrees of each vertex in $\tilde{\mathbf{A}}$, $\mathbf{W}^{(l)}$ is the trainable weight matrix, and σ is an activation function. $\mathbf{X}^{(l)}$ is the matrix of the hidden representations of the vertices at the l-th layer. $\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$ is called the renormalised adjacency matrix. We refer the reader to [Kipf and Welling, 2017] for the formal derivation.

Although GCNs may be interpreted as spatial-based models because they propagate messages along the edges in a graph, they still rely on an approximation of the graph Laplacian. We consider all models which utilise the graph Laplacian in any way, shape, or form as spectral-based. This allows us to maintain a strong distinction with spatial-based graph neural networks which explicitly perform convolutions using neighbouring vertices.

Spatial-based Approaches

Spatial-based graph neural networks formulate convolutions based on the neighbourhood around each vertex. We discuss Message Passing Neural Networks [Gilmer et al., 2017] and Graph Networks [Battaglia et al., 2018], two frameworks which generalise several existing GNNs.

Message Passing Neural Networks (MPNNs) is a unified framework which generalises both spatial-based and spectral-based GNNs [Gilmer et al., 2017]. MPNNs

are designed primarily to express spatial-based GNNs. Let G be an undirected graph with vertex features x_v and edge features e_{vw}. MPNNs consist of two phases: a message passing phase and a readout phase. The message passing phase is defined in terms of a shared message function M_t and a shared vertex update function U_t. In each message passing step, the new latent representation h_v^{t+1} for each vertex in the graph is computed based on the message m_v^{t+1}:

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw})$$
$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1})$$

where $N(v)$ denotes the neighbouring vertices of v in the graph G. Evidently, it is possible to repeatedly apply the message passing phase to continuously update the latent representations of the vertices in the graph. In every step of message passing, each vertex sends a 'signal' to its neighbouring vertices. Subsequently, repeated applications of message passing allow information to travel long distances in the graph.

The readout phase of MPNNs computes a graph-level output $\hat{y}$ using a readout function R which accepts the latent features of the vertices h_v^T after the last message passing step at time step T as input:

$$\hat{y} = R(\{h_v^T \mid v \in G\}).$$

The message function M^t, vertex update function U^t, and readout function R are all functions which are learned by a MPNN. That is, each function could be implemented as any learning model such as a MLP. Gilmer et al. [2017] show that MPNNs subsume notable GNNs including Convolutional Networks for Learning Molecular Fingerprints [Duvenaud et al., 2015], Gated Graph Neural Networks [Li et al., 2015], and Kipf and Welling [2017]'s spectral-based Graph Convolutional Networks.

Graph Networks (GNs) is a framework which subsumes an increased family of GNNs including MPNNs. GNs use flexible graph-to-graph modules which perform computations over the features and structures within a graph. These modules, which are called GN blocks, may be composed sequentially and repeatedly applied. Chapter 3 will present Hypergraph Networks, our extension of Graph Networks to *hypergraphs*, in detail.

Comparison between Spectral and Spatial GNNs

We break down the differences between spectral and spatial GNNs into two aspects: generality and flexibility [Wu et al., 2019].

Spectral GNNs generally assume that the graph is fixed, as the eigenvectors and eigenvalues of the normalised graph Laplacian may vary substantially across graphs. Although Kipf and Welling [2017]'s GCNs are able to generalise across graphs because they use localised first-order approximations of spectral graph convolutions, their generalisation capability is still limited in comparison to the capability of spatial

GNNs. Spatial GNNs are naturally able to generalise to graphs of varying structure and with different numbers of vertices and edges. This is because they perform convolutions on each vertex in a graph based on a vertex's local neighbourhood.

Directed graphs must be converted to undirected graphs before they may be applied to a spectral GNN, as the graph Laplacian is not well-defined for directed graphs. Moreover, spectral-based GNNs are only able to perform convolutions based on vertex-level features and do not support edge-level or global features as input. This is in contrast to spatial based GNNs including Graph Networks, which are able to support edge-level, vertex-level and global input features.

As we have demonstrated, spectral-based GNNs have significant limitations. For this reason, this thesis focuses on spatial-based approaches. Chapter 3 will introduce Hypergraph Networks, our generalisation of Graph Networks to *hypergraphs*.

2.3 Learning for Planning

Machine learning (ML) and deep learning (DL) approaches to planning have recently seen a significant increase in interest, given the ever-increasing ability of learning algorithms to automatically learn complex relationships and features from experience. Current ML and DL approaches for planning can be split into three major categories: planner selection, learning generalised policies, and learning heuristics.

Planner Selection

Planner selection involves selecting which planner out of a portfolio of planners should be applied for a given task. This is motivated by the fact that the performance of planners may vary across different tasks. Sievers et al. [2019] introduce Delfi, an online portfolio-based planner which firstly converts a planning task into an image by using the binary image representation of the adjacency matrix given by the *abstract structure graph* of the task, and then train Convolutional Neural Networks to learn which planner to invoke. Similarly, Ma et al. [2019] use Graph Neural Networks directly on the *problem description graph* and the *abstract structure graph* of a task for online planner selection and *adaptive scheduling*. We do not describe planner selection in detail as it is not particularly relevant for this thesis.

Learning Generalised Policies

We consider a stochastic policy as a function which returns a probability distribution over the actions an agent may apply in a given state. A generalised policy is a stochastic policy which may be applied across all problems in a given domain.

Groshev et al. [2018] use hand-crafted translators to convert states in a domain into a representation which may be used to train a Convolutional Neural Network (CNN) or Graph Convolutional Network (GCN) for learning generalised policies and heuristics. For example, states in Sokoban are converted into their corresponding grid-based image representation which contains the layout of the warehouse, location

of the boxes, initial state, goal state, etc. (see Figure 5.9 for an example) before they are passed to a deep CNN. Hence, the major limitation of Groshev et al. [2018]'s approach is that each domain requires hand-engineered translators.

Action Schema Networks (ASNets) [Toyer et al., 2019] define a dedicated neural network architecture which exploits the relational structure of planning problems encoded in (P)PDDL[1] [Younes and Littman, 2004] to learn generalised policies for deterministic and probabilistic planning. In contrast to [Groshev et al., 2018], ASNets do not require any hand-crafted input features or translators as the network design is automatically inferred from the PPDDL problem [Toyer et al., 2019]. An ASNet is composed of alternating action layers and proposition layers which are sparsely connected according to the structure of the action schemas defined in a PPDDL problem. One significant advantage of ASNets is its sophisticated weight sharing scheme which allows a network to theoretically generalise to problems of any size in a given domain. A disadvantage of ASNets is its fixed receptive field which limits its capability to support long chains of reasoning.

Issakkimuthu et al. [2018] defined custom neural network architectures which are able to learn policies for **individual** planning problems defined in RDDL [Sanner, 2011]. On the other hand, ToRPIDo [Bajpai et al., 2018] and TraPSNet [Garg et al., 2019] learn generalised policies which are able to transfer between RDDL problems, albeit with the assumptions of unary actions and binary non-fluents they impose on the problems. In contrast to ASNets which use a dedicated neural network architecture, ToRPIDo and TraPSNet use standard graph convolutional networks and graph attention networks [Veličković et al., 2018] to encode states in latent space, respectively.

Learning Heuristics

Although a copious amount of research exists for learning heuristics with deep learning for classic NP-hard combinatorial optimisation problems such as the Travelling Salesman and Knapsack problem [Khalil et al., 2017; Li et al., 2018], it is unclear how we may apply these techniques to exploit the relational structure of a planning problem. Nevertheless, there has been extensive research on how we may learn heuristics for planning using standard machine learning techniques including multilayer perceptrons (MLPs) and bootstrap-learning.

Yoon et al. [2008] learn linear heuristic functions which are weighted linear combinations of features derived from a relaxed plan. These features may include the length of the relaxed plan and information regarding the delete effects which were ignored by the relaxed plan. The heuristic functions which are learned by [Yoon et al., 2008] are domain-dependent, as they learn knowledge specific to the planning domain they were trained on.

Arfaee et al. [2010] and Geissmann [2015] use bootstrap-learning to iteratively learn a stronger heuristic function starting from a weak heuristic function. Their approach relies on a bootstrapping technique which adaptively updates the training

[1] PPDDL is an extension of PDDL which supports probabilistic planning.

data depending on whether the heuristic at a given iteration is able to successfully solve the training tasks. If the heuristic is able to solve a training task, the resulting state sequences for the plan are added to the training set. If a heuristic is unable to solve a training task, then states encountered on a random walk from the goal state are added to the training set – these represent states which are hopefully easier to solve. The updated training set is then used to train a stronger heuristic at the next iteration. Both works used a shallow MLP with a single hidden layer to learn a heuristic function over the training data. Geissmann [2015] shows that it is possible to learn a domain-independent heuristic for classical planning using bootstrap-learning.

Garrett et al. [2016] use Support Vector Machines (SVMs), a kernel-based machine learning algorithm [Boser et al., 1992], for learning domain-dependent heuristics which accurately ranks states as opposed to estimating the optimal heuristic h^*. The input features to a SVM for each pair of actions (a_1, a_2) include: the set intersection of the preconditions and effects for a_1 and a_2, the *temporal ordering* of a_1 and a_2 in an *approximate* plan, and features derived from an existing domain-independent heuristic including the heuristic value. One limitation of the heuristics learned by [Garrett et al., 2016] is that they can only be applied to search algorithms which operate on ranks, such as greedy best-first search. In contrast, the heuristics we learn may be applied to any search algorithm which uses the heuristic values, such as A*.

Gomoluch et al. [2017] learn domain-independent planning heuristics by training a MLP on features derived from h^{FF} [Hoffmann, 2001], including the number of operators in the FF relaxed plan, and the heuristic value. This can be considered as learning an improvement on h^{FF}. In contrast to [Garrett et al., 2016; Gomoluch et al., 2017; Yoon et al., 2008], our neural networks learn heuristics from scratch and do not use input features computed from existing domain-independent heuristics.

Chapter 3

Hypergraph Networks

In Section 2.2.3, we argued for the importance of a neural network architecture with a strong relational inductive bias, as it allows a network to exploit the inherent structure of the problem at hand. For example, Convolutional Neural Networks impose locality and translation invariance – biases which are effective for processing images as there is high covariance within the pixels in a local neighbourhood of an image.

Although there are existing neural network architectures designed for learning over hypergraphs, including Hypergraph Neural Networks [Feng et al., 2019] and HyperGCN [Yadati et al., 2018], deep learning on hypergraphs is still in its infancy in comparison to learning on standard graphs. Moreover, existing techniques rely predominantly on *spectral* rather than *spatial* convolutions.

In this chapter, we will rigorously define hypergraphs in Section 3.1 and then explore existing deep learning models in Section 3.2. We finish this chapter by presenting **Hypergraph Networks** (Section 3.3), our novel framework based on Graph Networks [Battaglia et al., 2018] which generalises and extends existing *spectral*-based and *spatial*-based hypergraph models. As we will discuss, the Hypergraph Networks framework exploits the inherent relational structure of a hypergraph and provides a powerful toolkit for constructing new learning models.

3.1 Hypergraphs

A hypergraph is a generalisation of a normal graph in which a hyperedge may connect any number of vertices together. A undirected hypergraph may be formulated as a pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, \ldots, v_{N^v}\}$ is the set of vertices of cardinality N^v (vertex set), and $\mathcal{E} = \{e_1, e_2, \ldots, e_{N^e}\}$ is the set of hyperedges of cardinality N^e with $e_i \subseteq \mathcal{V}$ (hyperedge set) [Gallo et al., 1993].

A hyperedge e is defined as the set of vertices which the hyperedge contains. Clearly, when $|e_i| = 2$ for $i \in \{1, \ldots, N^e\}$, then $\mathcal{H}$ is a standard undirected graph.

The incidence matrix $\mathbf{H}$ of a hypergraph is a $N^v \times N^e$ matrix where for $i \in \{1, \ldots, N^v\}$ and $j \in \{1, \ldots, N^e\}$:

$$\mathbf{H}[i, j] = \begin{cases} 1, & \text{if } v_i \in e_j \\ 0, & \text{otherwise.} \end{cases} \tag{3.1}$$

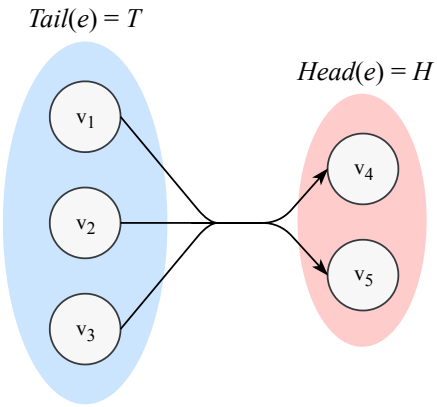


Figure 3.1: Example of a directed hyperedge $e = (T, H)$ where $Tail(e) = T = \{v_1, v_2, v_3\}$ and $Head(e) = H = \{v_4, v_5\}$.

Let $h(v, e)$ represent the entry in the incidence matrix $\mathbf{H}$ for vertex v and hyperedge e. A weighted undirected hypergraph is defined as a triple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$, where each hyperedge $e \in \mathcal{E}$ is associated with a weight $w(e) \in \mathbb{R}^+$.

Directed Hypergraphs

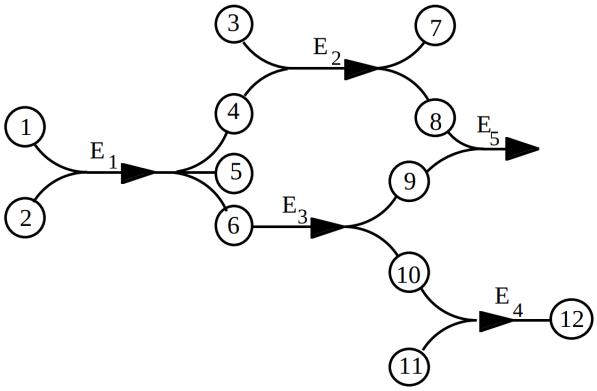
A directed hypergraph is a hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with directed hyperedges. A directed hyperedge $e \in \mathcal{E}$ is a pair (T, H) where T and H are vertex sets. Let $Tail(e) = T \subseteq V$ be the tail, and $Head(e) = H \subseteq V$ be the head of e, respectively. An example of a directed hyperedge is shown in Figure 3.1. The incidence matrix $\mathbf{H}$ of a directed hypergraph is a $N^v \times N^e$ matrix where for $i \in \{1, \ldots, N^v\}$ and $j \in \{1, \ldots, N^e\}$:

$$\mathbf{H}[i, j] = \begin{cases} -1, & \text{if } v_i \in Tail(e_j) \\ 1, & \text{if } v_i \in Head(e_j) \\ 0, & \text{otherwise.} \end{cases} \tag{3.2}$$

Figure 3.2 depicts an example of a directed hypergraph and its corresponding incidence matrix. A weighted directed hypergraph is a triple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$, where each hyperedge $e \in \mathcal{E}$ is directed and associated with a weight $w(e) \in \mathbb{R}^+$.

3.2 Deep Learning on Hypergraphs

Research on deep learning for hypergraphs is still in its infancy, despite the influx of new models for regular graphs. Here, we will present and compare the current state-of-the-art hypergraph learning algorithms. We will discuss these models in a fair amount of detail as our framework, Hypergraph Networks, may be used to define all of them.



	E_1	E_2	E_3	E_4	E_5
1	-1	0	0	0	0
2	-1	0	0	0	0
3	0	-1	0	0	0
4	1	-1	0	0	0
5	1	0	0	0	0
6	1	0	-1	0	0
7	0	1	0	0	0
8	0	1	0	0	-1
9	0	0	1	0	-1
10	0	0	1	-1	0
11	0	0	0	-1	0
12	0	0	0	1	0

Figure 3.2: Example of a directed hypergraph and its corresponding incidence matrix [Gallo et al., 1993]. Note that the hyperedge E_5 has an empty head.

3.2.1 Hypergraph Neural Networks

Hypergraph Neural Networks (HGNNs) [Feng et al., 2019] is the first work to define *spectral* hypergraph convolutions by approximating hypergraph Laplacians in terms of first-order Chebyshev polynomials. The hypergraph Laplacian is a generalisation of the graph Laplacian discussed in Section 2.2.4. Spectral convolutions rely on the formulation of the graph in the spectral (Fourier) domain. HGNNs are able to learn the higher-order relationships between the entities in the data, and were shown to outperform existing state-of-the-art methods that do not exploit the structure of hypergraphs.

Formulation

Given an undirected weighted hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$, we define $\mathbf{W} \in \mathbb{R}^{N^e \times N^e}$ and $\mathbf{D}_e \in \mathbb{R}^{N^e \times N^e}$ to be the diagonal matrices with each diagonal entry containing the weight $w(e_i)$ and degree $\delta(e_i) = \sum_{v \in \mathcal{V}} h(v, e_i)$ for the hyperedge e_i for $i \in \{1, \dots, N^v\}$, respectively. Furthermore, $\mathbf{D}_v \in \mathbb{R}^{N^v \times N^v}$ is defined as the diagonal matrix representing the degree of each vertex with each diagonal entry containing $d(v_j) = \sum_{e \in \mathcal{E}} w(e) h(v_j, e)$ for $j \in \{1, \dots, N^v\}$.

We refer the reader to [Feng et al., 2019] for full details on how the hyperedge convolution is derived in terms of Fourier transforms, Chebyshev polynomials, and relevant approximations and simplifications. A full treatment of this derivation is not presented in this thesis as the derivation is based heavily on spectral graph theory. A hyperedge convolutional layer in a HGNN is defined as:

$$\mathbf{X}^{(l+1)} = \sigma(\mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{D}_v^{-1/2}\mathbf{X}^{(l)}\mathbf{\Theta}^{(l)}) \tag{3.3}$$

where $\mathbf{X}^{(l)} \in \mathbb{R}^{N^v \times C_{(l)}}$ is the signal (i.e., features) of the hypergraph at the l-th layer,

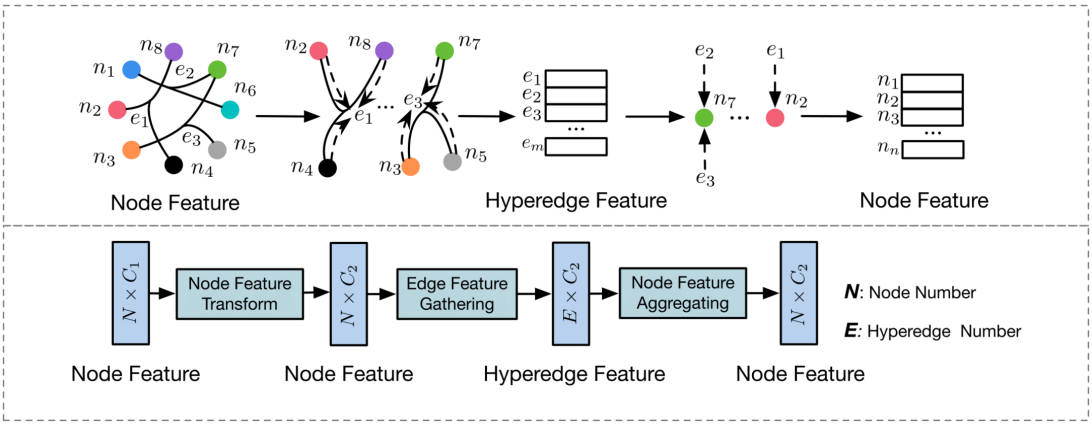


Figure 3.3: Illustration of a single hyperedge convolutional layer from Figure 4 of [Feng et al., 2019].

$\mathbf{\Theta}^{(l)} \in \mathbb{R}^{C_{(l)} \times C_{(l+1)}}$ are the weights to be learned, $\mathbf{X}^{(l+1)} \in \mathbb{R}^{N^v \times C_{(l+1)}}$ is the output of the hyperedge convolutional layer, and σ is a non-linearity activation function (e.g. sigmoid, ReLU [Nair and Hinton, 2010]). $C_{(l)}$ and $C_{(l+1)}$ represents the number of channels (i.e., dimensionality) for the vertex features in the l-th and $(l+1)$-th layer, respectively.

$\mathbf{L} = \mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{D}_v^{-1/2}$ is called the normalised incidence matrix, as it combines information about the degree of each vertex, the degree and weight of each hyperedge, and the structure of the hypergraph. If we look closely at the matrix multiplication between $\mathbf{L}$ and $\mathbf{X}$ in Equation 3.3, we can observe that a new feature for each vertex v is computed as a weighted aggregation of its neighbouring vertices (as determined by the hyperedges that contain v). We refer the reader to Section 10.1 in the Appendix of [Gilmer et al., 2017] for a formal derivation of this reasoning applied to a Kipf and Welling [2017]'s standard Graph Convolutional Network.

It is important to note that this formulation of a hyperedge convolutional layer is only defined in terms of undirected hypergraphs. However, our experiments showed that using HGNNs with directed hypergraphs converted to undirected hypergraphs yielded admirable results.

The input to the first hyperedge convolutional layer are the initial vertex features $\mathbf{X}^{(1)} \in \mathbb{R}^{N^v \times d}$, where d is the initial dimensionality for each vertex feature. Hence, a disadvantage of HGNNs is that they do not support hyperedge features as explicit input to the network. However, a single HGNN layer performs vertex-edge-vertex transforms which subsequently computes intermediate hidden hyperedge representations (Figure 3.3).

[Bai et al., 2019] extend HGNNs by introducing a hypergraph attention module which enhances the representational capability of a network. The attention mechanism is used to directly update the incidence matrix, such that each entry now measures the degree of connectivity between a vertex and a hyperedge rather than a binary value representing whether a vertex is in a hyperedge or not. Despite being more expensive to train, the hypergraph attention module can be used to a "learn a

dynamic connection of hyperedges" which can lead to richer feature embeddings.

Pitfalls

A hyperedge convolutional layer returns vertex-level outputs, and hence cannot be used to make edge-level predictions. However, the vertex-level outputs can be aggregated to produce a single hypergraph-level output (e.g., by doing a summation of the node features output by the last layer of the HGNN). That being said, HGNNs are primarily used to solve semi-supervised node classification problems.

As we have previously discussed in Section 2.2.4, the major disadvantage of spectral-based graph neural networks is that they usually generalise poorly to hypergraphs with a different number of vertices and hyperedges to the ones the network was trained on. Moreover, spectral-based approaches are inefficient as the convolutions are performed over the whole hypergraph rather than a batch of vertices [Wu et al., 2019].

Moreover, hyperedge convolutional layers in a HGNN rely on a first-order Chebyshev polynomial approximation of the spectral graph convolution, where the eigenvectors of the hypergraph Laplacian essentially act as the Fourier bases and the eigenvalues act as frequencies [Feng et al., 2019]. Not surprisingly, the eigenvectors and eigenvalues of the hypergraph Laplacian can vary significantly due to perturbations to the hypergraph structure. Hence, we would expect the generalisation performance of a HGNN to be limited.

3.2.2 HyperGCN

HyperGCN [Yadati et al., 2018] is a spectral-based approach for learning over hypergraphs that relies on decomposing the hypergraph into a standard graph. It does this by "approximating each hyperedge by a set of pairwise edges connecting the vertices of the hyperedge", and then applying a conventional spectral-based Graph Convolutional Network (GCN) [Kipf and Welling, 2017]. As with HGNNs, HyperGCNs are only defined for undirected hypergraphs.

Formulation

HyperGCNs first construct a standard weighted graph G_S on the vertex set $\mathcal{V}$ of a weighted undirected hypergraph $(\mathcal{V}, \mathcal{E}, w)$ by using Chan et al. [2018]'s definition of the Hypergraph Laplacian. The hypergraph Laplacian is a generalisation of the graph laplacian introduced in Section 2.2.4.

In a HyperGCN, G_S is constructed by adding standard edges $\{i_e, j_e\} \subseteq e \colon e \in \mathcal{E}$ with weights $w(\{i_e, j_e\}) = w(e)$. Recall a hyperedge e is defined as a vertex set. Hence, a HyperGCN initially decomposes the hypergraph into a standard graph, where each vertex in a hyperedge e is connected to the other vertices in e through standard edges.

Then, for each hyperedge e at a given epoch τ, HyperGCN selects the single representative standard edge (i_e, j_e) where the hidden feature representations of i_e and j_e differ the most. This procedure is called the hypergraph Laplacian. Recall that

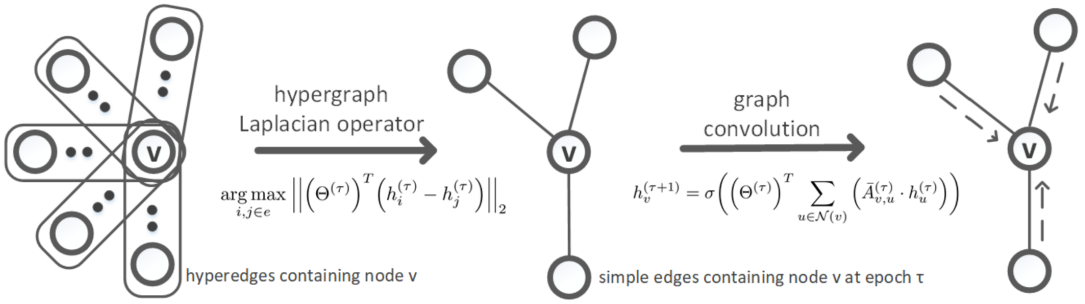


Figure 3.4: Illustration of a single convolution on a vertex v using HyperGCN for epoch τ (Figure 1 in [Yadati et al., 2018]). Θ is a trainable weight matrix, $\overline{A}$ is the normalised adjacency matrix of the standard graph obtained from decomposing the hypergraph, and h_i and h_j are the hidden representations of the vertices i_e and i_j, respectively. For each hyperedge e at a given epoch τ, HyperGCN selects the standard edge where the hidden representations of the vertices differ the most, as defined by the equation given in the 'hypergraph Laplacian operator' step. HyperGCN then applies a standard Graph Convolutional layer over the resulting graph.

an epoch refers to a single pass of the data set and the associated weight updates to the HyperGCN. Hence, the representative edges which are selected by a HyperGCN may vary across epochs as the hidden representations of the vertices evolve over time.

Now, HyperGCN applies a standard GCN over the resulting graph. These steps are depicted in Figure 3.4. Evidently, the selected edge may not be representative of the hyperedge which it was constructed from, and hence we may expect the representational capability of a HyperGCN to be limited in comparison to that of HGNNs.

Yadati et al. [2018] address this issue by introducing "mediators", where for the single representative edge (i_e, j_e) for each hyperedge e, new edges $\{(i_e, k) : k \in e \text{ s.t. } k \neq i_e \wedge k \neq j_e\}$ and $\{(j_e, k) : k \in e \text{ s.t. } k \neq i_e \wedge k \neq j_e\}$ are connected and added to the resulting graph to be processed by the GCN. Essentially, each representative edge (i_e, j_e) is now conditioned on the other vertices in the hyperedge e.

Comparison to HGNNs and Pitfalls

In comparison to HGNNs, which approximate each hyperedge with a clique and hence require a polynomial number of edges, HyperGCNs only require a linear number of edges as they select one representative standard edge for each hyperedge. This leads to faster training time, at the potential loss of representational capability. However, Yadati et al. [2018] found that HyperGCNs outperformed HGNNs for semi-supervised node classification and combinatorial optimisation, most likely due to the removal of noisy hyperedges as a result of this linear approximation.

HyperGCNs and HGNNs both suffer from the issue that their convolutions are formulated in terms of the spectral-domain of the graph. This means that they usually generalise poorly to hypergraphs with a different number of vertices and hyperedges to the ones they were trained on, for the reasons aforementioned at the end of Section

3.2.1. Moreover, both techniques are only defined for undirected hypergraphs, and cannot be used to make edge-level predictions as their layers only give vertex-level outputs.

Despite this, Yadati et al. [2018] found that HyperGCNs were able to generalise to different sized hypergraphs for the densest k-subhypergraph problem, an NP-hard problem. Our experiments found that spatial-based hypergraph networks are able to generalise far better than spectral-based hypergraph networks in terms of planning performance.

3.2.3 Dynamic Hypergraph Neural Networks

To the best of our knowledge, Dynamic Hypergraph Neural Networks (DHGNN) [Jiang et al., 2019] is the first hypergraph deep learning framework which utilises *spatial* convolutions. A DHGNN is composed of stacked layers, where each layer consists of a dynamic hypergraph construction (DHG) module and hypergraph convolution (HGC) module. DHGNNs are only defined for undirected hypergraphs. However, unlike HGNNs and HyperGCNs, it is possible to extend DHGNNs to directed hypergraphs.

In contrast to a spectral convolution which is defined in the spectral domain of the entire hypergraph, a spatial convolution is defined in terms of the local neighbourhoods of vertices and hyperedges (i.e., batches of vertices and hyperedges). This allows a DHGNN to generalise to hypergraphs with a different number of vertices and hyperedges to the hypergraph(s) it was trained on.

The DHGNN framework is most similar to the Hypergraph Networks framework we will introduce in Section 3.3, than to HGNNs and HyperGCNs. Next, we will explain the formulation of DHGNNs in more depth. It is important to note that DHGNN was published in parallel to the development of this thesis.

Dynamic Hypergraph Construction (DHG) Module

The DHG module allows the undirected hypergraph structure to be dynamically refined as the feature embeddings of the vertices evolve over time, as the initial hypergraph may not have been the most suitable representation of the data. Ideally, DHG will lead to a hypergraph structure that better models the higher-order relationships in the data.

DHG is most applicable to datasets where the hypergraph structure must be inferred from the data, rather than being explicitly specified. Because of this, DHG is not applicable for learning heuristics over hypergraphs, as the hypergraph representation we utilise is explicitly specified by the grounded planning problem. Moreover, as we will discuss in Chapter 4, learning heuristic over hypergraphs involves estimating shortest paths rather than explicitly learning feature embeddings.

Let $Con(e) = \{v_1, \ldots, v_{k_e}\}$ and $Adj(v) = \{e_1, \ldots, e_{k_v}\}$ represent the k_e vertices a hyperedge e contains (vertex set) and the set of hyperedges that contain vertex v (adjacent hyperedge set), respectively. Moreover, let $k = |Con(e)| = k_e$ be the size of

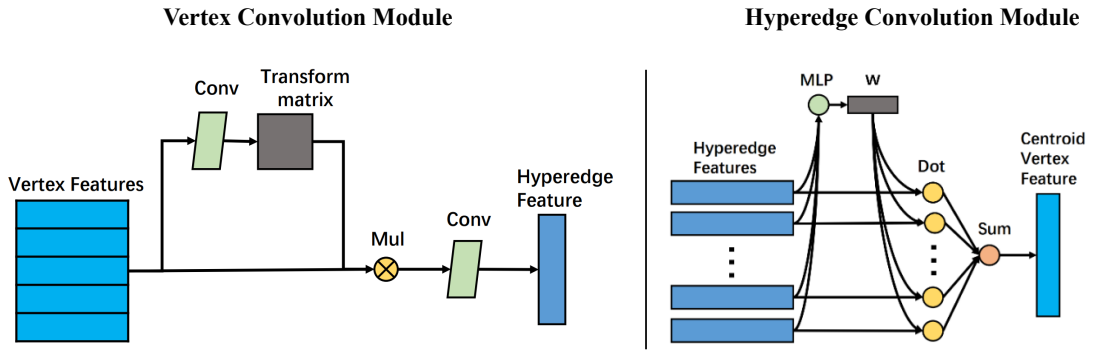


Figure 3.5: The Vertex Convolution module (left) and Hyperedge Convolution module (right) of a DHGNN. Taken from Figure 3 and 4 of [Jiang et al., 2019].

a new hyperedge e, and $S = |Adj(v)| = k_v$ be the size of the new adjacent hyperedge set of a vertex v in the new hypergraph constructed by DHG.

DHG firstly uses k-nearest neighbours to compute the $k-1$ nearest neighbours for each vertex v. These neighbouring vertices, along with v are used to construct a new hyperedge for v in $Adj(v)$ which is clearly of size k. Then, DHG uses k-means clustering to compute n clusters over all the vertex features (where $n > S-1$ is a hyperparameter). For each vertex v, the $S-1$ nearest clusters will then be assigned as adjacent hyperedges of this vertex, and are subsequently added to $Adj(v)$. The hyperedge for each cluster is constructed by computing the $k-1$ nearest vertices to the cluster centre and appending the current vertex v – clearly the resulting hyperedge contains k vertices. Evidently, DHG is used to dynamically update the hypergraph structure by exploiting local structures (through k-nearest neighbours) and global structures (through k-means clustering). We refer the reader to Algorithm 1 in [Jiang et al., 2019] for more details.

Hypergraph Convolution (HGC) Module

A HGC module consists of the vertex convolution submodule and the hyperedge convolution submodule, both depicted in Figure 3.5. Vertex convolution is used to aggregate vertex features to a hyperedge, while hyperedge convolution is used to aggregate hyperedge features to a vertex.

Recall that the DHG module dynamically updates the structure of the hypergraph such that each hyperedge contains k vertices. The vertex convolution module learns a transform matrix $\mathbf{T}$ of size $k \times k$, which is used along with a 1-dimensional convolution operator to transform and compact the k vertex features into a new hyperedge feature (left diagram in Figure 3.5). Vertex convolution is used to compute the new feature $\mathbf{x}_e$ for each hyperedge $e \in \mathcal{E}$ in the hypergraph by enabling "inter-vertex and inter-channel information" flow.

In terms of implementing vertex convolution, a Multilayer Perceptron (MLP) is used to generate this transform matrix $\mathbf{T}$ from the sampled vertex features $\mathbf{X}_v$ (concatenated vertex features) for a hyperedge e, i.e. $\mathbf{T} = MLP_1(\mathbf{X}_v)$. Then, a 1-

dimensional convolution is applied to the result of the transform matrix multiplied by the hidden representation of the vertex features in order to get the new hyperedge feature, $\mathbf{x}_e$. That is, $\mathbf{x}_e = conv\_1d(\mathbf{T} \cdot MLP_2(\mathbf{X}_v))$ where MLP_2 is a MLP used to compute the hidden representation of the vertex features $\mathbf{X}_v$

The hyperedge convolution module employs an attention mechanism, in which the new feature $\mathbf{x}_v$ of a vertex v is computed as a weighted sum of the adjacent hyperedge features $\mathbf{X}_e$ (i.e., the concatenated features of the hyperedges that contain v). Each weight of a hyperedge can be viewed as the 'importance' that hyperedge has to contributing to the vertex feature (right diagram in Figure 3.5). Hyperedge convolution is applied to compute a new feature $\mathbf{x}_v$ for each vertex $v \in \mathcal{V}$ in the hypergraph.

In terms of implementation, a MLP with a Softmax is used to compute the normalised weight scores $\mathbf{w}$ for the S adjacent hyperedges, i.e. $\mathbf{w} = softmax(MLP(\mathbf{X}_e))$. These weights are then used to compute the new feature $\mathbf{x}_v$ for a vertex v, by computing a weighted sum over the features for each hyperedge $\mathbf{x}_e$. That is, $\mathbf{x}_v = \mathbf{w}^T\mathbf{X}_e$.

For more details regarding the HGC module, we refer to reader to Algorithm 2 in [Jiang et al., 2019]. It is evident that HGCs, and hence DHGNNs, are defined in terms of spatial convolutions not spectral convolutions, and hence are expected to have better generalisation performance.

Pitfalls

A DHGNN layer, as defined by Jiang et al. [2019], only returns vertex-level outputs. This can be attributed to the fact that DHGNNs are designed mainly for performing semi-supervised node classification. However, DHGNNs are extensible to support hyperedge-level outputs by storing the explicit results of the vertex convolution module. This is in contrast to HGNNs and HyperGCNs which cannot practically support hyperedge-level outputs as they operate on the incidence matrix of a hypergraph.

A disadvantage of the vertex convolution module is that it only uses the vertex set features to compute a new hyperedge's feature, and hence ignores the hyperedge's current embedding. Similarly, the hyperedge convolution module only uses the adjacent hyperedge features to compute a new vertex feature, and hence ignores the current embedding of the vertex.

Once again, these disadvantages can be attributed to the fact that for most semi-supervised node classification tasks, the hypergraph structure must be inferred from the data rather than being explicitly specified, and hence the structure may be dynamically updated. Thus, DHGNNs are likely to be unsuitable for the majority of combinatorial optimisation problems, where the hypergraph structure is explicitly defined (e.g., computing shortest paths over hypergraphs) and hyperedge-level outputs may be required.

3.2.4 Other Related Work

Learning over hypergraphs was first formally introduced in the seminal work [Zhou et al., 2007], where the authors introduced spectral hypergraph clustering and embedding techniques using the hypergraph Laplacian. However, this approach is computationally inefficient in comparison to HGNN and HyperGCN, which adopt a more efficient Chebyshev expansion of this Laplacian and an approximation of a hyperedge by a set of pairwise edges, respectively.

Xu et al. [2018] view ordered binary decision diagrams as 3-uniform hypergraphs, where each hyperedge must contain exactly 3 vertices. They then decompose 3-uniform hypergraph message passing into two ordinary message passing steps in a Message Passing Neural Network (MPNN) [Gilmer et al., 2017], which is designed for ordinary graphs. Their results show that the resulting MPNNs can find near optimal solutions in a short amount of time.

Yadati et al. [2019] propose Neural Hyperlink Predictors (NHP), which are based on Graph Convolutional Networks (GCNs), for hyperlink prediction. Hyperlink prediction is concerned with predicting the probability of a future possible link (i.e., a new hyperedge). NHPs use the clique expansion of the dual of a hypergraph to approximate it as a planar graph, and then use conventional GCNs.

3.3 Hypergraph Networks (HGNs)

As briefly mentioned in Section 2.2.4, Graph Networks (GN) [Battaglia et al., 2018] is a framework capable of defining and extending the majority of existing graph neural networks. We propose Hypergraph Networks, our generalisation of Graph Networks to hypergraphs. Since hypergraphs are generalisations of standard graphs, all the models that may be represented using Graph Networks may also be represented using Hypergraph Networks.

Hypergraph Networks can be used to represent both spectral-based and spatial-based hypergraph neural network models, but are focused on representing the latter. HGNs may be considered as message-passing models, where information flows between the hyperedges and vertices based on the structure of a hypergraph in each message-passing step.

Hypergraph Networks (HGNs) are designed to have extremely flexible building blocks which exploit the relational structure of hypergraphs. In the subsections that follow, we will define the representation of a hypergraph in a HGN, the HGN block, and argue for the relational inductive biases that rise from HGNs.

3.3.1 Hypergraph Representation

Unlike HGNNs, HyperGCNs and DHGNNs which were presented in Section 3.2, the Hypergraph Networks framework is explicitly designed for directed hypergraphs. It is easy to extend HGNs to undirected hypergraphs. From this point onwards,

we assume that a hypergraph is a directed hypergraph. When a hypergraph is not directed, we will explicitly state that it is an undirected hypergraph.

The following hypergraph representation is a modification of the representation for standard graphs presented in [Battaglia et al., 2018]. As such, we preserve the naming of most variables. A hypergraph in the HGN framework is a triple $G = (\mathbf{u}, V, E)$ where:

- $\mathbf{u}$ represents the global attributes of the hypergraph (i.e., hypergraph-level attributes).
- $V = \{\mathbf{v}_i : i \in \{1, \dots, N^v\}\}$ is the set of N^v vertices, where $\mathbf{v}_i$ represents the i-th vertex's attributes.
- $E = \{(\mathbf{e}_k, R_k, S_k) : k \in \{1, \dots, N^e\}\}$ is the set of N^e hyperedges, where:
  - $\mathbf{e}_k$ represents the k-th hyperedge's attributes,
  - $R_k \subseteq V$ 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
  - $S_k \subseteq V$ is the vertex set which contains the indices of the vertices which are in the tail of the k-th hyperedge (i.e. senders).

An example of a hypergraph in the HGN framework representation is depicted in Figure 3.6.

Note, that the size of R_k and S_k varies depending on the number of vertices in the head and tail of the k-th hyperedge. This representation is in contrast to Graph Networks, where receivers and senders are only defined for a single vertex. Consequently, all graphs that may be defined using the graph representation in Graph Networks, may also be defined in our hypergraph representation. We can simply set $R_k = \{r_k\}$ and $S_k = \{s_k\}$ for the k-th edge, where r_k is the index of the receiver vertex, and s_k is the index of the sender vertex.

3.3.2 Hypergraph Network (HGN) Block

A Hypergraph Network block is a hypergraph-to-hypergraph function which forms the core building block of a HGN. The internal structure of a full HGN block is identical to that of a GN block [Battaglia et al., 2018], except now the edge update function ϕ^e must supports multiple receivers and senders. HGN blocks are designed to be configurable to the task at hand, whether it be semi-supervised node classification or combinatorial optimisation.

A full HGN 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 $\rho^{v\to u}$:

$$\begin{aligned} \mathbf{e}'_k &= \phi^e(\mathbf{e}_k, \mathbf{R}_k, \mathbf{S}_k, \mathbf{u}) & \overline{\mathbf{e}}'_i &= \rho^{e\to v}(E'_i) \\ \mathbf{v}'_i &= \phi^v(\overline{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}) & \overline{\mathbf{e}}' &= \rho^{e\to u}(E') \\ \mathbf{u}' &= \phi^u(\overline{\mathbf{e}}', \overline{\mathbf{v}}', \mathbf{u}) & \overline{\mathbf{v}}' &= \rho^{v\to u}(V') \end{aligned} \tag{3.4}$$

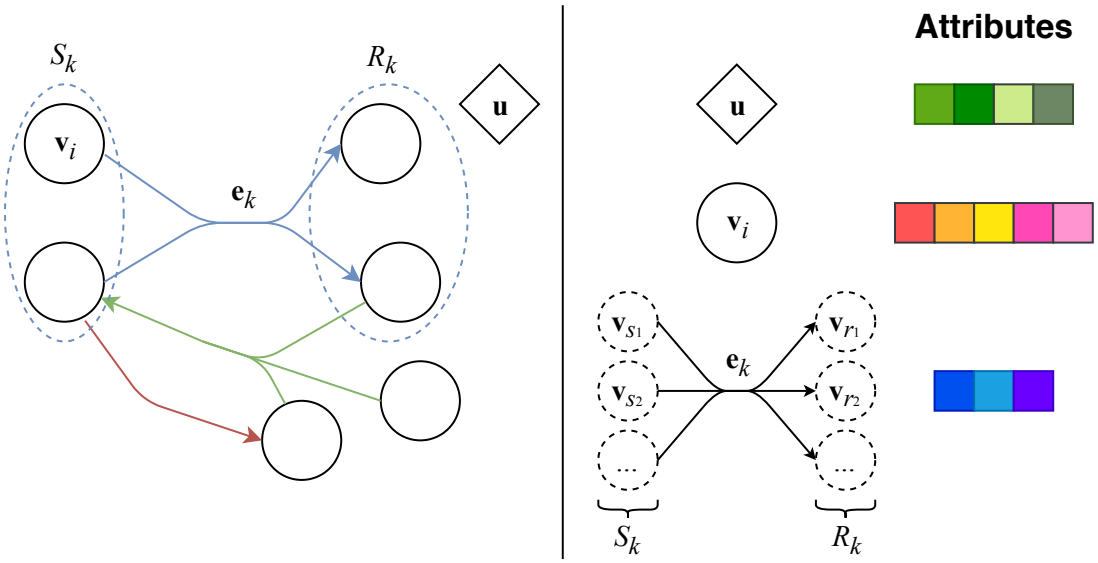


Figure 3.6: Example of a Hypergraph represented in a Hypergraph Network (adaptation of Figure 2 from Battaglia et al. [2018]). The attributes are properties of the entity it represents, and could be encoded as vectors, matrices, sets, etc.

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 $E'_i = \{(\mathbf{e}'_k, R_k, S_k) : k \in \{1, \dots, N^e\} \text{ s.t. } i \in R_k\}$, $V' = \{\mathbf{v}'_i : i \in \{1, \dots, N^e\}\}$, and $E' = \bigcup_i E'_i = \{(\mathbf{e}'_k, R_k, S_k) : k \in \{1, \dots, N^e\}\}$. Essentially, E'_i represents the hyperedges where the i-th vertex is a receiver vertex (i.e., in the head of the hyperedge), E' represents all the hyperedges, and V' represents all the 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

The hyperedge update function ϕ^e computes the updated hyperedge attribute $\mathbf{e}'_k$ using the current hyperedge's attributes, the attributes of the receiver and sender vertices, and the global attributes. The vertex update function ϕ^v computes the updated vertex attribute $\mathbf{v}'_i$ using the aggregated information from all the hyperedges it 'receives' a signal from (i.e., it appears in the head of the hyperedge), the current vertex's attributes, and the global attributes. Finally, the global update function ϕ^u computes the updated global attributes using the aggregated information from all the hyperedges and vertices in the hypergraph, along with the current global attributes.

In a single pass of a HGN block, the hyperedge update function ϕ^e is applied to all the hyperedges to compute per-hyperedge updates, then the vertex update function ϕ^v is applied to all the vertices to compute per-vertex updates, and lastly, the global update function ϕ^u is applied once to compute the new global attributes [Battaglia

Algorithm 1 Computation steps in a full HGN block. Adapted from Algorithm 1 in [Battaglia et al., 2018].

function HYPERGRAPHNETWORKBLOCK($\mathbf{u}, V, E$)
  for $k \in \{1, \dots, N^e\}$ **do**
    ▷ 1. Compute updated hyperedge attributes
    $\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \{\mathbf{v}_j : j \in R_k\}, \{\mathbf{v}_j : j \in S_k\}, \mathbf{u})$
  for $i \in \{1, \dots, N^v\}$ **do**
    ▷ 2. Aggregate hyperedge attributes for vertex
    $E'_i \leftarrow \{(\mathbf{e}'_k, R_k, S_k) : k \in \{1, \dots, N^e\} \text{ s.t. } k \in R_k\}$
    $\overline{\mathbf{e}}'_i \leftarrow \rho^{e \to v}(E'_i)$
    ▷ 3. Compute updated vertex attributes
    $\mathbf{v}'_i \leftarrow \phi^v(\overline{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$
  ▷ 4. Aggregate hyperedge attributes globally
  $E' \leftarrow \{(\mathbf{e}'_k, R_k, S_k) : k \in \{1, \dots, N^e\}\}$
  $\overline{\mathbf{e}}' \leftarrow \rho^{e \to u}(E')$
  ▷ 5. Aggregate vertex attributes globally
  $V' \leftarrow \{\mathbf{v}'_i : i \in \{1, \dots, N^e\}\}$
  $\overline{\mathbf{v}}' \leftarrow \rho^{v \to u}(V')$
  ▷ 6. Compute updated global attribute
  $\mathbf{u}' \leftarrow \phi^u(\overline{\mathbf{e}}', \overline{\mathbf{v}}', \mathbf{u})$
  return $(\mathbf{u}', V', E')$

et al., 2018]. Algorithm 1 describes these updates in more detail.

In contrast to the edge update function ϕ^e in a GN block, ϕ^e in a HGN block should be able to compute an update based on a variable number of sender and receiver vertex attributes, even if it does not use all these attributes. For example, the ϕ^e update in a HGN may be implemented as a function which firstly samples n sender vertex features from $\mathbf{S}_k$, and m receiver vertices $\mathbf{R}_k$ (to ensure fixed vector sizes), and then applies a standard neural network such as a MLP over the resulting features.

3.3.3 Relational Inductive Biases and Combinatorial Generalisation

Recall from Section 2.2.3 that relational inductive biases are the inductive biases which impose constraints on the relationships and interactions of the entities in a learning algorithm [Battaglia et al., 2018].

Hypergraph Networks impose a stronger form of relational inductive biases than Graph Networks, as hyperedges allow us to model interactions between multiple entities using a single relationship. In contrast, Graph Networks only allow us to model interactions between exactly two entities in a single relationship through a standard edge. Thus, hypergraphs allow us to express arbitrary relationships between entities, while graphs only allow us to express **pairwise** arbitrary relationships between entities [Battaglia et al., 2018].

Additionally, the input to a Hypergraph Network (i.e. the hypergraph structure

itself) determines how the entities interact, not the architecture of the framework. A hyperedge connecting a set of entities indicates that the entities should interact, while the absence of a hyperedge between a set of entities indicate that they are isolated from each other and hence should not interact [Battaglia et al., 2018].

HGNs are invariant to permutations in the ordering of vertices and hyperedges in a hypergraph, as these entities are expressed as sets within the framework. Of course, the invariance of a specific instance of a HGN relies on how the update functions are implemented. For example, if the implementation of an update function assumes that there is a specific ordering in the vertex or hyperedge features it receives, then the resulting HGN will not be invariant.

The inherent design of HGNs means that they are able to support combinatorial generalisation, since they apply per-hyperedge and per-vertex updates across all hyperedges and vertices in the hypergraph, respectively. This means that a HGN can scale to hypergraphs with different structures and different numbers of vertices and hyperedges.

3.3.4 Configurable HGN Blocks and Existing Models as HGNs

In this subsection, we discuss how the flexible internal structure of a HGN block may be modified and configured to suit the task at hand, by depicting how the existing deep learning hypergraph models presented in Section 3.2 may be defined using HGNs. Figure 3.7 depicts a full Hypergraph Network block (defined in Section 3.3.2), where all hypergraph attributes, update functions, and aggregation functions are utilised.

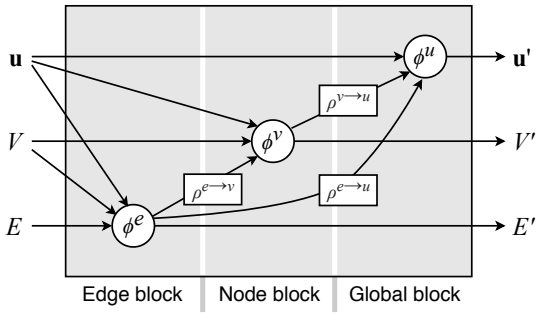


Figure 3.7: The full Hypergraph Network block configuration which predicts global, vertex and hyperedge outputs based on the incoming global, vertex and hyperedge attributes [Battaglia et al., 2018]. The incoming arrows to an update function ϕ represent the inputs it receives.

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 3.7) is a function $f\colon (\mathbf{e}_k, \mathbf{R}_k, \mathbf{S}_k, \mathbf{u}) \mapsto \mathbf{e}'_k$ 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.

Composable HGN Blocks

Since the input and output of a HGN block is a hypergraph, an arbitrary number HGN blocks can be composed sequentially by passing the output of one block as the input to another. These blocks can either be unshared (each block contains different update and aggregation functions), or shared (the same block is reused) [Battaglia et al., 2018]. Multiple compositions of the same shared block may be interpreted as message-passing [Gilmer et al., 2017], where the identical update and aggregation functions of the block are applied iteratively to incrementally propagate information through the hypergraph. This is akin to belief-propagation in Bayesian Networks.

Hypergraph Neural Networks as HGNs

Recall a hyperedge convolutional layer in a HGNN is defined as (Section 3.2.1):

$$\mathbf{X}^{(l+1)} = \sigma(\mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{D}_v^{-1/2}\mathbf{X}^{(l)}\mathbf{\Theta}^{(l)}),$$

and that HGNNs only support undirected hypergraphs. Let us assume the vertex set for each hyperedge is encoded in its receiver vertex set (i.e. the sender vertex set is empty). Now, we can use the senders vertex set R_k of each hyperedge in $\{(\mathbf{e}_k, R_k, \emptyset) : k \in \{1, \ldots, N^e\}\}$ to derive the incidence matrix $\mathbf{H}$, vertex degree matrix $\mathbf{D}_v$, and the hyperedge degree matrix $\mathbf{D}_e$. The weight of each hyperedge can be encoded in its feature $\mathbf{e}_k$, and then extracted to get the weight matrix $\mathbf{W}$. However, since all these matrices stay fixed for a given hypergraph, we only need to compute $\mathbf{L} = \mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{D}_v^{-1/2}$ once. Thus, the input edge attributes are not strictly required as input to the block, as we can encode $\mathbf{L}$ in the global attributes $\mathbf{u}$ of each hypergraph, i.e., let $\mathbf{u} = \mathbf{L}$.

A hyperedge convolutional layer at the l-th layer can be defined as the HGN block shown in Figure 3.8. Without a loss in generality, we assume that the vertex update function computes the update for all vertices $V = \{\mathbf{v}_i : i \in \{1, \ldots, N^v\}\}$ simultaneously (i.e., ϕ^v accepts multiple vertices as input, and outputs multiple vertices). Then we, define:

$$\begin{aligned}
\phi^v &\coloneqq f^v(V, \mathbf{u}) \\
&= \sigma\left(\mathbf{u}\left[\mathbf{v}_1, \ldots, \mathbf{v}_{N^v}\right]^T \Theta^{(l)}\right) \\
&= \sigma\left(\mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{D}_v^{-1/2}\mathbf{X}^{(l)}\Theta^{(l)}\right),
\end{aligned}$$

where σ is the activation function, $\mathbf{X}^{(l)} = \left[\mathbf{v}_1, \ldots, \mathbf{v}_{N^v}\right]^T$ is the matrix containing the features of the input vertices V, and $\Theta^{(l)}$ is the learnable weight matrix. We use $[\mathbf{x}, \mathbf{y}, \mathbf{z}]$ to refer to the concatenation of vectors $\mathbf{x}, \mathbf{y},$ and $\mathbf{z}$.

It is possible to show that the matrix multiplication between $\mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{D}_v^{-1/2}$ and $\mathbf{X}$ computes a new feature for each vertex v by taking a weighted aggregation of its neighbours, and hence ϕ^v may be defined to compute per-vertex updates. However, for the sake of brevity, we do not derive this relationship as it is not the main

focus for this thesis. We refer the reader to Section 10.1 in the Appendix of [Gilmer et al., 2017] for a formal derivation of this reasoning applied to a Kipf and Welling [2017]'s standard Graph Convolutional Network.

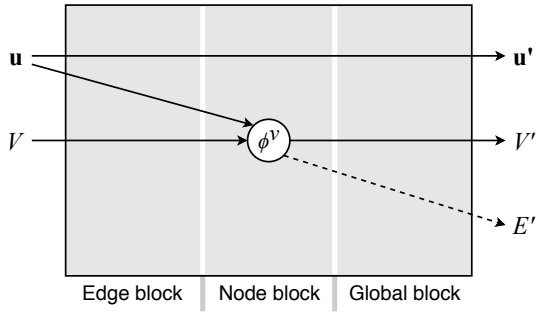


Figure 3.8: A HGNN hyperedge convolutional layer represented as a HGN block. The dotted line to E' represents the hidden hyperedge representation which is implicitly computed by the layer's vertex-hyperedge-vertex transform (Figure 3.3). As the matrix $\mathbf{L} = \mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{D}_v^{-1/2}$ stays constant for a given hypergraph, it can be stored in the global attributes $\mathbf{u}$ of the hypergraph and propagated unmodified.

HyperGCN as HGNs

HyperGCNs learn functions over hypergraphs by decomposing the hypergraph into a standard graph by using the hypergraph Laplacian, and then applying a spectral-based Graph Convolutional Network. We investigate how 1-HyperGCN, which selects one representative standard edge for each hyperedge in the hypergraph, may be represented using HGNs (see Section 3.2.2).

We consider the convolutional layer depicted in Figure 3.4, which may be represented in HGNs using the block configurations shown in Figure 3.9. The hyperedge update function ϕ^e in the hypergraph Laplacian HGN block is implemented as:

$$\begin{aligned}\phi^e &\coloneqq f^e(\mathbf{R}_k, \emptyset) \\ &= \mathit{create\_edge}\left(\arg\max_{\mathbf{v}_i,\mathbf{v}_j \in \mathbf{R}_k} ||\Theta^T(\mathbf{v}_i - \mathbf{v}_j)||_2\right)\end{aligned}$$

where Θ is the matrix of weights which need to be learned. Since HyperGCNs are only defined for undirected hypergraphs, we assume that the vertex set for the k-th hyperedge is encoded in the receivers $\mathbf{R}_k$ (i.e., the senders $\mathbf{S}_k$ are empty). It is important to note that E and E' are used for the sole purpose of the representing the (hyper)graph structure, as (Hyper)GCNs only support undirected (hyper)graphs.

The GCN block computes a new feature for each vertex in the standard graph whose structure is encoded E' (Figure 3.9). The vertex update function ϕ^v is defined

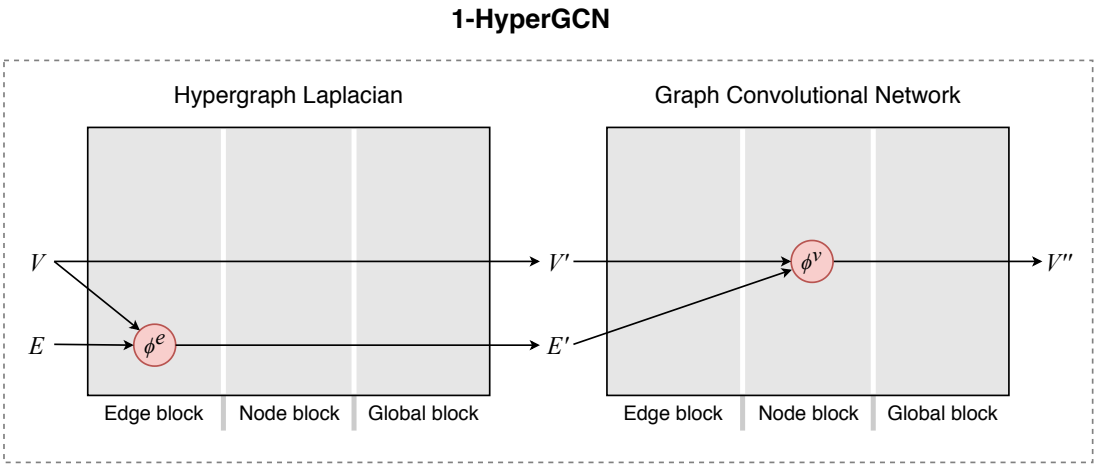


Figure 3.9: A graph convolution layer in a 1-HyperGCN. The hypergraph Laplacian block converts the hypergraph into a standard graph by selecting the "most representative" pair of vertices for each hyperedge. The Graph Convolutional Network computes a new feature for each vertex in this standard graph. The red update functions across the two blocks indicate that they share the same weights Θ.

as:

$$\begin{aligned}\phi^v &:= f^v(\mathbf{v}'_i, E') \\ &= \sigma\left(\Theta^T \sum_{v_j \in \mathcal{N}(v_i)} (\tilde{A}[i,j] \cdot \mathbf{v}'_i)\right)\end{aligned}$$

where σ is a non-linear activation function, $\mathcal{N}(v_i)$ is a function which returns the neighbours for v_i (derived from E'), and $\tilde{A}$ is the normalised adjacency matrix of the graph (derived from E'). Critically, Θ is the **same** weight matrix used in the hypergraph Laplacian HGN block. We refer the reader to [Yadati et al., 2018] for more details regarding this procedure.

Dynamic Hypergraph Neural Networks as HGNs

A DHGNN layer is made up of the Dynamic Hypergraph Construction (DHG) module, and the Hypergraph Convolution (HGC) module. We consider how to represent the HGC module using HGNs, as they define per-vertex and per-hyperedge updates. In contrast, the DHG module reconstructs the hypergraph using k-Means clustering and k-nearest neighbours – this procedure is not relevant for this thesis. Figure 3.10 depicts a Hypergraph Convolution (HGC) module.

We define the hyperedge update function ϕ^e in the vertex convolution block as:

$$\begin{aligned}\phi^e &:= f^e(\mathbf{R}_k, \emptyset) \\ &= conv\_1d\left(MLP_1(\mathbf{R_k}) \cdot MLP_2(\mathbf{R}_k)\right)\end{aligned}$$

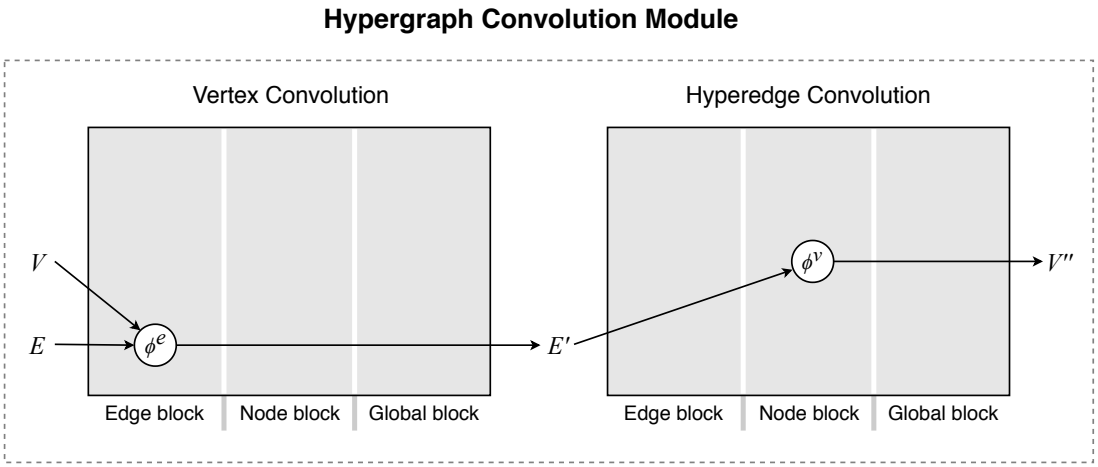


Figure 3.10: A DHGNN hypergraph convolution module represented as two sequential HGN blocks for vertex convolution and hyperedge convolution. The vertex convolution block uses the vertex features and structure of the hypergraph to compute new hyperedge features. The hyperedge convolution block uses the hyperedges and structure of the hypergraph to compute new vertex features.

where $\mathbf{R_k} = [\mathbf{v}_1, \ldots, \mathbf{v}_k]^T$ is the matrix of features for the k vertices in the current k-th hyperedge's vertex set. Since DHGNNs are only defined for undirected hypergraphs, we assume that the vertex set for the k-th hyperedge is encoded in the receivers $\mathbf{R}_k$ (i.e., the senders $\mathbf{S}_k$ are empty). MLP_1 is a multi-layer perceptron used to generate the transform matrix for $\mathbf{R}_k$, and MLP_2 is used to compute the hidden representation of $\mathbf{R}_k$. ϕ^e is applied to each hyperedge in the hypergraph to compute and output a new hyperedge feature.

We define the vertex update function ϕ^v in the hyperedge convolution block as:

$$\begin{aligned}\phi^v &\coloneqq f^v(\mathbf{E}'_i) \\ &= softmax(MLP(\mathbf{E}'_i))^T \mathbf{E}'_i\end{aligned}$$

where $\mathbf{E}'_i = [\mathbf{e}'_1, \ldots, \mathbf{e}'_S]$ are the concatenated features of the S hyperedges in the i-th vertex's adjacent hyperedge set derived from the output E' of the vertex convolution block (see Figure 3.10). MLP is used to compute the self-attention weights, and $softmax$ is used to normalise the weights to 1. $softmax(MLP(\mathbf{E}_i))$ is a vector containing the normalised weights for each hyperedge, which are used to compute the new vertex feature by performing a weighted sum over the input hyperedge features $\mathbf{E}'_i$. ϕ^v is applied to each vertex in the hypergraph in order to compute and output a new vertex feature.

Thus, we have shown that it is possible to define the Hypergraph Convolution modules in a DHGNN as two sequential HGN blocks.

3.3.5 Summary

We have introduced Hypergraph Networks, our framework which generalises Graph Networks [Battaglia et al., 2018] to hypergraphs. We have shown that HGN blocks are extremely powerful hypergraph-to-hypergraph functions with highly flexible within-block designs. HGNs may be easily configured to support an array of tasks, including vertex-level classification, hyperedge-level classification, and combinatorial optimisation.

We argued that HGNs impose a stronger form of relational inductive biases than Graph Networks, as hypergraphs can model complex interactions between multiple entities using a single relationship. On the other hand, graphs are only able to model pairwise interactions between two entities using a standard edge. Moreover, HGNs are able to support combinatorial generalisation to hypergraphs with different structures and different numbers of vertices and hyperedges, as they apply shared per-hyperedge and per-vertex updates over all hyperedges and vertices in a hypergraph, respectively.

Finally, we showed that it is possible to model existing of hypergraph deep learning algorithms as HGNs. In Section 4.2.2 of the next chapter, we present our HGN architecture which is used to learn heuristics over hypergraphs for planning.

Chapter 4

Learning Heuristics over Hypergraphs

In Chapter 3, we introduced Hypergraph Networks, a powerful framework for building deep learning models which operate over hypergraphs. In this chapter, Section 4.1 firstly discusses how the computation of delete-relaxation heuristics can be expressed as approximations of shortest paths over hypergraphs.

Following this, Section 4.2 will present STRIPS-HGNs, our recurrent *encode-process-decode* Hypergraph Network architecture which we use to learn heuristics by approximating shortest paths in latent space. Moreover, we investigate how the input and output features for a planning problem may be modelled as HGN hypergraphs, describe the within-block design of our architecture, argue for the combinatorial generalisation of a STRIPS-HGN, and discuss its limitations. Finally, Section 4.3 frames learning a heuristic as a regression problem, and presents the training algorithm we use to generate training samples and optimise the weights of the update functions in a STRIPS-HGN by using gradient descent.

4.1 Delete-Relaxation Heuristics as Shortest Paths over Hypergraphs

Recall the delete relaxation of a STRIPS problem $P = \langle F, O, I, G, c \rangle$, is the STRIPS problem $P^+ = \langle F, O', I, G, c \rangle$, where $O' = \{\langle Pre(o), Add(o), \emptyset \rangle \mid o \in O\}$ (as defined in Section 2.1.1).

Hypergraph Generation

We generate the weighted directed hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$, induced by P^+ using the procedure described in Algorithm 2. Essentially, each proposition in F represents a vertex in $\mathcal{V}$, and each grounded action $o \in O'$ represents a hyperedge $e \in \mathcal{E}$ where $Tail(e) = Pre(o)$ and $Head(e) = Add(o)$. An example of the hyperedge for an action is depicted in Figure 4.1.

Algorithm 2 Steps for generating the weighted directed hypergraph $(\mathcal{V}, \mathcal{E}, w)$ from the delete relaxed STRIPS problem, P^+.

function GENERATEHYPERGRAPH($P^+ = \langle F, O', I, G, c \rangle$)
  $\mathcal{V} \leftarrow F$ ▷ The vertices of the hypergraph are all the propositions
  $\mathcal{E} \leftarrow \{\}$
  $w \leftarrow$ an empty map
  ▷ Construct a directed hyperedge $e = (Tail(e), Head(e))$ for each action
  for $o \in O'$ **do**
    $\mathcal{E} \leftarrow \mathcal{E} \cup (Pre(o), Add(o))$
    $w(o) \leftarrow c(o)$ ▷ the weight of the hyperedge is the action's cost
  return $(\mathcal{V}, \mathcal{E}, w)$

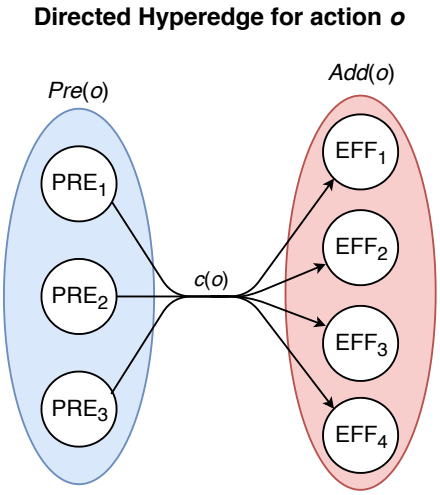


Figure 4.1: Example of the hyperedge generated by Algorithm 2 for an action o with 3 preconditions and 4 positive effects, and a cost $c(o) = 1$.

4.1.1 h^{max} and h^{add} as shortest paths over hypergraphs

Let $g_s(\omega)$ represent the minimum cost required to achieve proposition ω (atom) from the current state s [Bonet and Geffner, 2001]. $g_s(\omega)$ is derived by computing the fixed-point of Equation 4.1, where initially, $g_s(\omega) = 0$ if $w \in s$ and $g_s(\omega) = +\infty$ otherwise.

$$\begin{aligned} g_s(\omega) &= \min_{o \in O:\ \omega \in Add(o)} \{g_s(\omega), c(o) + g_s(Pre(o))\} \\ &= \min_{e \in \mathcal{E}:\ \omega \in Head(e)} \{g_s(\omega), w(e) + g_s(Tail(e))\} \end{aligned} \tag{4.1}$$

where $g_s(Pre(o))$ and $g_s(Tail(e))$ represents the cost of achieving the sets of propositions given by $Pre(o)$ and $Tail(e)$, respectively. Equation 4.1 essentially computes the minimum cost of achieving a proposition ω, by choosing an action o that adds ω and minimises the cost of applying o plus the cost of achieving the preconditions of o.

We follow Haslum and Geffner [2000]'s notation, and define the cost of achieving a set of propositions (vertices), $\Delta \subseteq F \equiv \mathcal{V}$, in the state s as:

$$g_s(\Delta) = \oplus_{\omega \in \Delta}\, g_s(\omega) \tag{4.2}$$

where $\oplus$ is an aggregation operator which aggregates the costs of achieving each proposition $\omega \in \Delta$. We can now define a heuristic $h^{\oplus}(s) = g_s(G)$ for the state s, where G are the goal propositions.

When $\oplus = \sum$, then $h^{\oplus}(s) = h^{add}(s)$. Evidently, h^{add} approximates the cost of achieving a set of propositions, either the goal set or the preconditions of actions, as the cost of achieving each proposition in the set independently of the others. This independence assumption simplifies the computation but does not reflect reality, since making a proposition ω true might have the side effect of making other propositions $w' \in \Delta$ also true. Hence, h^{add} is not admissible as it may overestimate the true cost of achieving the propositions in Δ. Nevertheless, h^{add} is informative as the computed heuristic values provide some information about the cost of achieving each goal proposition.

When $\oplus = \max$, then $h^{\oplus}(s) = h^{max}(s)$. It is easy to see that the cost of achieving the most difficult proposition in a set, either the goal set or the preconditions of actions, cannot be greater than the cost of achieving all the propositions in Δ. Thus, h^{max} is admissible. However, because h^{max} ignores the costs of achieving the other goal propositions and other preconditions, it is not as informative as h^{add}.

Relation to Hypergraphs

The weighted directed hypergraph induced by P^+ encapsulates the propositions and actions required to compute $h^{\oplus}$. Although this hypergraph representation is not explicitly required for computing $h^{\oplus}$, we theorise that a deep neural network will be able to learn a better function $\oplus$ to aggregate the proposition and action features over the hypergraph structure, in a rich latent feature space.

It is important to note that an efficient implementation of h^{max} and h^{add} incrementally generates the hypergraph when computing a heuristic value (in fact, this generation is implicit), whereas our Hypergraph Networks require the entire hypergraph to passed as input to the network for a given state.

Steinmetz and Torralba [2019] presented the first work to formally bridge the gap between several classes of planning heuristics and hypergraphs. The authors defined *hyperabstractions*, which are heuristics based on the fundamental ideas behind abstraction and critical-path heuristics. We refer the reader to their paper, which was published in parallel to the development of this thesis, for further details.

4.2 STRIPS-HGNs: a Hypergraph Network for Learning Heuristics

We denote the architecture of our HGNs for learning heuristics over hypergraphs as STRIPS-HGNs, which we abbreviate to STRIPS-HGN. STRIPS-HGNs are designed to be highly adaptable to different input features for each proposition and action, as well as being agnostic to the exact implementation of each update function in its HGN blocks. This gives us the power to adjust the internal network implementations based on our desired application.

4.2.1 STRIPS-HGN Hypergraph Representation

Recall that in Section 3.3, we defined a hypergraph under the HGN framework as a triple $G = (\mathbf{u}, V, E)$. Moreover, we discussed the composability of HGN blocks, which are hypergraph-to-hypergraph functions.

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:

1. $\mathbf{u}_{\text{inp}} = \emptyset$, as global features are not required as input to a STRIPS-HGN. Nevertheless, it is easy to adapt STRIPS-HGNs to support global features. For example, we could supplement a STRIPS-HGN with a heuristic value $h(s)$ computed by a heuristic h for the state s, such that the network learns an 'improvement' on h. This idea is similar to [Gomoluch et al., 2017], where the authors learn improvements on h^{FF} using a simple MLP.
2. $V_{\text{inp}} = \{\mathbf{v}_i : i \in \{1, \ldots, |F|\}\}$ contains the input features (as a vector) for the $|F|$ propositions in the problem. Features for a proposition could include whether the proposition is true for the current state or goal state, and whether the proposition is a *fact landmark* computed by Landmark Count for the state s [Richter and Westphal, 2010].
3. $E_{\text{inp}} = \{(\mathbf{e}_k, R_k, S_k) : k \in \{1, \ldots, |O'|\}\}$ 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 action o's input features (as a vector), which could include the cost of the action $c(o)$, and whether the action is in the *disjunctive action landmarks* computed by LM-cut for the state s.
   - $R_k = Add(o)$ is the vertex set containing the indices of the vertices in the additive effects of o.
   - $S_k = Pre(o)$ is the vertex set containing the indices of the vertices in the 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 with the heuristic value for state s, $V_{\text{out}} = \emptyset$, and $E_{\text{out}} = \emptyset$. As the output of a planning heuristic is defined to be a single real number, it is unnecessary to output any features for V_{out} and E_{out}.

Input Hypergraph Construction Algorithm

Let VertexFeature be the function which maps a vertex to a feature based on the STRIPS problem P and the current state s. Similarly, let HyperedgeFeature be the function that maps a hyperedge e to a feature based on the STRIPS problem P, the current state s, and the cost of the action $w(e)$. GenerateHGNHypergraph in Algorithm 3 describes the formal procedure required to generate an input HGN hypergraph G.

Algorithm 3 Generating the input hypergraph required by a STRIPS-HGN based on the STRIPS problem P, and the current state s.

function GenerateHGNHypergraph(P, s)
  ▷ 1. Generate delete-relaxation hypergraph in the mathematical sense
  $P^+ \leftarrow$ DeleteRelax(P) ▷ Get delete relaxed STRIPS problem
  $(\mathcal{V}, \mathcal{E}, w) \leftarrow$ GenerateHypergraph(P^+) ▷ Defined in Algorithm 2
  ▷ 2. Compute hypergraph in HGN representation
  ▷ Compute vertex and hyperedge features based on problem and current state
  $V \leftarrow \{\text{VertexFeature}(v, P, s) : v \in \mathcal{V}\}$
  $E \leftarrow \{(\text{HyperedgeFeature}(e, P, s, w(e)), \mathit{Head}(e), \mathit{Tail}(e)) : e \in \mathcal{E}\}$
  $G \leftarrow (\emptyset, V, E)$ ▷ Global attributes are undefined for STRIPS-HGNs
  return G

4.2.2 STRIPS-HGN 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], as depicted in Figure 4.2.

The input hypergraph G_{inp} is firstly encoded to 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 attributes 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 4.2). Evidently, this procedure results in $M - 1$ intermediate hypergraph outputs, one for

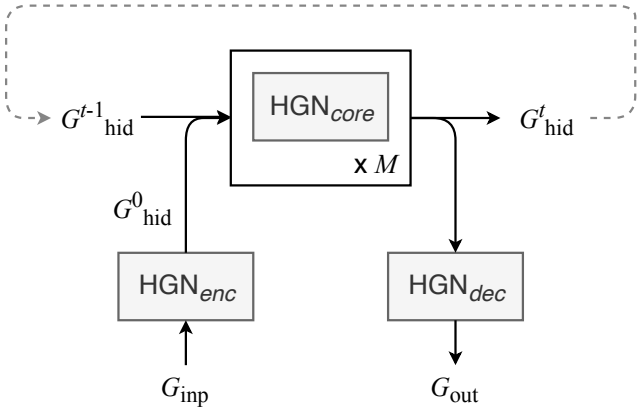


Figure 4.2: The architecture for a STRIPS-HGN, which uses a recurrent "Encode-process-decode" architecture (modified from Figure 6c in [Battaglia et al., 2018]). The merging line for G^0_{hid} and G^{t-1}_{hid} indicates concatenation, while the splitting lines that are output by the HGN$_{core}$ block indicates 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 $t+1$.

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 attribute $\mathbf{u}_{\text{out}}$. Observe that we can decode each latent hypergraph which is output by HGN$_{core}$ to obtain a heuristic value for each time step $t \in \{1, \ldots, M\}$. This fact will be exploited in our training algorithm, which is introduced in Section 4.3.

Core Block Details

We can interpret a STRIPS-HGN as a message passing model which performs M steps of message passing, as the shared processing block HGN$_{core}$ is repeated M times using a recurrent architecture. Although this means that a vertex only receives a 'signal' from other vertices at most M hops away[1], we theorise that this is sufficient to learn a powerful function which aggregates proposition and action features in latent space.

Of course, we can solve this issue by setting M to be an arbitrarily large. However, this would make the network expensive to evaluate and unfeasible to apply to planning. In practice, we found that setting $M = 10$ was sufficient to achieve promising results.

Gilmer et al. [2017] propose a potential solution to this problem by creating a new "master" vertex which is connected to every other vertex in the hypergraph with a special hyperedge type. This master vertex, which would have a very high feature dimensionality, would act as a global scratch space for vertices to read and write to

[1] A hop represents a single crossing of a hyperedge (disregarding a hyperedge's weight).

in each step of message passing. This would in theory, allow signals to "travel long distances" even for a small number of message passing steps [Gilmer et al., 2017]. We did not experiment this approach due to the higher costs of training and evaluation, and the difficulties we encountered implementing it. We leave the investigation of the master vertex design for future work.

In contrast to neural network architectures such as Action Schema Networks [Toyer et al., 2019] 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 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 4.3a), HGN_{enc} encodes the vertex and hyperedge input features independently of each other using its ϕ^v and ϕ^e update functions, respectively.

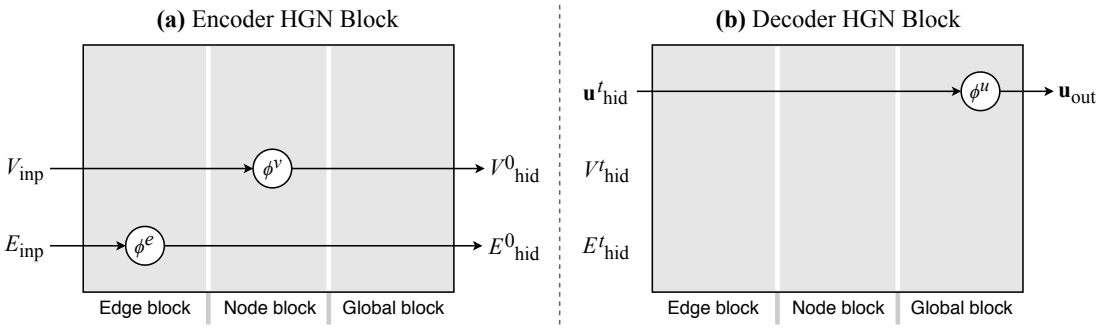


Figure 4.3: The Encoding and Decoding blocks of a STRIPS-HGN. An encoder block independently encodes the vertex and hyperedge features into latent space, while the decoder block decodes the latent global features into a single heuristic value. Figure 4.2 shows how these blocks are used by a STRIPS-HGN in relation to the core processing block.

The core processing block of a STRIPS-HGN (Figure 4.4) takes the concatenated vertex and hyperedge features from the latent hypergraphs G^0_{hid} and G^{t-1}_{hid} as input. The hyperedge update function ϕ^e computes per-hyperedge updates based on these hyperedge and vertex features. The vertex update function 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, the global update function 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.

Evidently, the global features from the previous timestep are not used by the HGN_{core} block. We adopt this design as we believe the latent representation of the heuristic value should be computed based solely on the vertex and hyperedge features, which incrementally become more informative as we perform more rounds

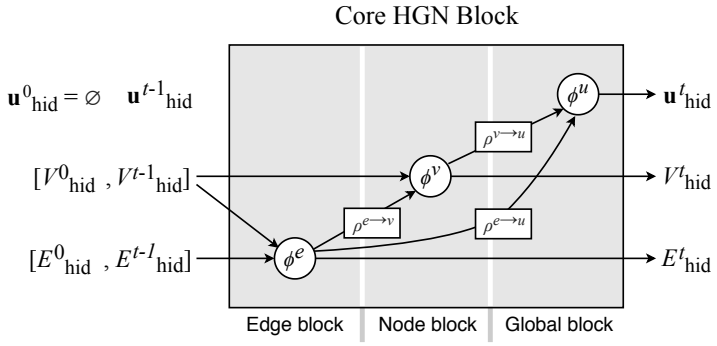


Figure 4.4: The Core processing block of a STRIPS-HGN, which computes per-hyperedge and per-vertex updates using the concatenated input hypergraph. The processing block additionally computes the global attribute $\mathbf{u}^t_{\text{hid}}$ using the aggregated vertex and hyperedge features. $\mathbf{u}^t_{\text{hid}}$ represents the latent features for the heuristic value. The global attribute $\mathbf{u}^{t-1}_{\text{hid}}$ computed in the previous time step is not used by the core block. $[x, y]$ refers to the concatenation of x and y. Figure 4.2 depicts how the core block is used by a STRIPS-HGN in relation to the encoder and decoder blocks.

of message passing. Additionally, the global features are undefined for the initial latent hypergraph G^0_{hid}.

The decoder block (Figure 4.3b) takes the latent representation of the global attributes $\mathbf{u}^t_{\text{hid}}$ of the hypergraph returned by the core HGN block, and uses the ϕ^u update function 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 for 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. We detail our choice of update and aggregation functions in Section 5.1.2, which describes our experimental setup.

Adapting STRIPS-HGNs to Learning Actions

Although STRIPS-HGNs are designed for learning heuristics, the flexibility of the HGN framework means that we can easily adapt them to learn which action to apply for a state s instead. We describe how this could be achieved.

We maintain the same input HGN hypergraph design as described in Section 4.2.1. However, we redefine the output hypergraph to be $G_{\text{out}} = (\mathbf{u}_{\text{out}}, V_{\text{out}}, E_{\text{out}})$ where $\mathbf{u}_{\text{out}} = \varnothing$, $V_{\text{out}} = \varnothing$, and $E_{\text{out}} = \{(\mathbf{e}^{\text{out}}_k, \varnothing, \varnothing) \colon k \in \{1, \dots, |O|\}\}$ with each $\mathbf{e}^{\text{out}}_k \in \mathbb{R}^{1\times 1}$ representing the probability of selecting the action represented by the hyperedge in state s.

The within-block design of the core block is updated to remove the need to output the global feature $\mathbf{u}^t_{\text{hid}}$, which was previously used to represent the latent representation of the heuristic value. The decoder block now decodes the hyperedge features

E^t_{hid} into a probability distribution which represents the confidence the network has in applying each action. Similar to Action Schema Networks (ASNets) [Toyer et al., 2019], the decoder block would apply a *masked softmax* activation function to remove actions which are not applicable for the state s (i.e. set them to have a probability of 0), and to normalise the probability distribution to 1.

However, as discussed in the motivation behind this thesis in Chapter 1, learning heuristics is a more robust approach than learning actions, as combining a heuristic with a search algorithm provides formal guarantees. Although it is possible to plug a hypervisor on top of a learning algorithm which outputs action probability distributions, as Shen et al. [2019] did with Monte-Carlo Tree Search and ASNets, learning heuristics generally provides a more efficient and lighter layer of reasoning to a search algorithm.

4.2.3 Combinatorial Generalisation

As briefly discussed in Section 3.3.5, the inherent design of a HGN supports combinatorial generalisation. Rather than applying a fixed transformation over the entire hypergraph, a STRIPS-HGN applies shared computations over each vertex through the update functions ϕ^v, and shared computations over each hyperedge through the update functions ϕ^e [Battaglia et al., 2018].

Because of this, we would expect that a STRIPS-HGN is able to generalise to planning problems (and hence hypergraphs) of different sizes to the ones it was trained on. Indeed, we found this was the case for several of the domains we experimented on, as we will present in our experimental results in Section 5.3.2.

In the case of training and testing a STRIPS-HGN on problems from a single domain, we theorise that the network is able to learn patterns in the latent representations of the vertices and hyperedges in local neighbourhoods of a hypergraph. This helps it determine how critical the actions in the neighbourhood may be in reaching the goal state, and subsequently how much of the 'signal' should be transmitted from this neighbourhood through message passing.

As we will show in Section 5.3.3, a single STRIPS-HGN may be trained on small problems from several domains, and is able to generalise to larger problems from the same collection of domains it was trained on. Moreover, it is possible for a single STRIPS-HGN to generalise to problems from different domains to the domains it was trained on. In the latter scenario, we believe a STRIPS-HGN provides a 'ranking' of which actions will lead us closer to the goal rather than providing accurate heuristic values, as the latent vertex and hyperedge features may not be extremely informative for the domains we did not train the network on.

4.2.4 Limitations of STRIPS-HGNs

One significant limitation of STRIPS-HGNs is that it is expensive to compute a single heuristic value for a state, given the cost of the high dimensional matrix operations required by each HGN block. Moreover, the computation cost of a STRIPS-HGN

scales with the size of the hypergraph, as messages need to be broadcasted to more vertices and hyperedges. However, if the heuristic learned by a STRIPS-HGN is very informative and provides estimates near the optimal heuristic h^*, then this cost would pay off if it reduces the total time required for a search algorithm to find a near-optimal solution in comparison to using h^{add}, h^{max}, and LM-cut.

The number of message passing steps M for the core HGN block is a hyperparameter which, in theory, should be selected based on how 'far' away the current state is from the goal. However, determining a good value for M this is not trivial, and would likely require computing an existing planning heuristic or a relaxed plan. This would further increase the computational cost required to obtain a heuristic estimate for a STRIPS-HGN. Ideally, we would want to embed a procedure into STRIPS-HGNs which is able to automatically select the number of message passing steps based on the convergence of the heuristic values output by the network at each time step.

Finally, we are unable to provide any 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.

4.3 Training Algorithm

Framing the Learning Problem

We frame learning a heuristic function h as a regression problem, where the heuristic function ideally provides near-optimal estimates of the cost to go. Although our experiments showed that the learned heuristics occasionally gave abnormal values when generalising to problems of significantly larger sizes than those a STRIPS-HGN was trained on, the heuristic values still provide enough information that A* finds a near-optimal plan. For example, we observed that when generalising to very large problems, $h(s) > 999999$ for all states s in a problem, almost as if the original heuristic was scaled by a constant factor.

Garrett et al. [2016] explicitly framed learning a heuristic as learning to rank states, as they focused on Greedy Best-First Search which is agnostic to the actual magnitude of the heuristic values. The authors argued that search performance is governed mostly by the ordering of the states induced by the heuristic, rather than the actual values of the heuristic. Although we observed that this was indeed true, we believe the values of the heuristic still provide important information which increases the probability of a search algorithm finding a near-optimal plan, potentially by providing some unprovable form of admissibility.

It is also possible to frame learning a heuristic as a classification problem, where a separate class is defined for each natural number up to a heuristic value limit. However, a classification-based approach would be more expensive to train given the higher-dimensionality output which is required. Moreover, the heuristic estimates are limited by the total number of classes, meaning that a classification-based approach

would be unable to generalise as well to larger problems compared to a regression-based approach.

4.3.1 Training Data Generation

We train our STRIPS-HGNs with the values generated by the optimal heuristic h^*. Algorithm 4 describes the training data generation procedure for a set of training problems $\{P_1, \ldots, P_n\}$.

Algorithm 4 Generating training data for a set of STRIPS problems $\{P_1, \ldots, P_n\}$.

function GENERATETRAININGDATA($\{P_1, \ldots, P_n\}$)
  $\mathcal{T} \leftarrow \{\}$ ▷ Set of tuples with (hypergraph, heuristic value)
  for $P_i \in \{P_1, \ldots, P_n\}$ **do**
    ▷ Compute the states of the optimal plan for the problem P
    $s_0, \ldots, s_N \leftarrow$ COMPUTEOPTIMALPLAN(P) ▷ s_0 is the initial state
    for $j \in \{0, \ldots, N\}$ **do**
      ▷ Append new training pair (hypergraph, heuristic value)
      $G \leftarrow$ GENERATEHGNHYPERGRAPH(P_i, s_j) ▷ Defined in Algorithm 3
      $\mathcal{T} \leftarrow \mathcal{T} \cup (G, N - j)$ ▷ $h^*(s_j) = N - j$ as we assume unit action costs
  return $\mathcal{T}$

For a set of planning problems, we firstly compute the optimal plan for each problem using any method (e.g., a dedicated solver for a domain, or A* with an admissible heuristic), and generate the trajectory of states $s_0, \ldots, s_N$ by executing the plan from the initial state s_0.

Recall that we assume actions have unit cost. Subsequently, the $N - j$ for each $s_j \in [s_0, \ldots, s_N]$ represents the optimal heuristic value $h^*(s_i)$. Although we assume unit costs, our training data generation algorithm may be easily extended to actions with different costs.

It is important to note that we can train STRIPS-HGNs to also learn h^{max} and h^{add}, with the subsequent learned heuristic achieving similar performance to the heuristic it was trained on. However, this is redundant as a STRIPS-HGN which is trained to learn the optimal heuristic h^* achieves significantly better planning performance than a network trained to mimic h^{max} and h^{add}.

4.3.2 STRIPS-HGN Weight Optimisation

After generating the training data $\mathcal{T}$, we use supervised learning to optimise a STRIPS-HGN. Algorithm 5 describes our training procedure, which we explain in detail below.

We assume that the implementation of each update function in the encoder, core, and decoder blocks of a STRIPS-HGN has some weights that need to be learned. For example, if we implemented each update function as a Multilayer Perceptron (MLP), the weights would be the weighted connections between each neuron in one layer and

Algorithm 5 Algorithm for optimising the weights θ of the update functions in a STRIPS-HGN.

```
1: procedure TRAIN-STRIPS-HGN(𝒯)
2:     θ ← INITIALISEWEIGHTS()
3:     epoch ← 0
4:     while epoch < MAX-EPOCHS and max training time not exceeded do
5:         ▷ Randomly sample minibatches of size BATCH-SIZE from 𝒯
6:         for ℬ ∈ SAMPLEMINIBATCHES(𝒯, BATCH-SIZE) do
7:             Update θ using dℒ_θ(ℬ)/dθ (Equation 4.3)
```

the neurons in the next layer (as depicted in Figure 2.1). For simplicity, we aggregate the weights of all update functions into a single variable θ.

Additionally, the weights may need to be initialised depending on the choice of model implementation. For a MLP, this would involve setting small random weights to ensure that a STRIPS-HGN does not get stuck in a local optima during training.

Since we framed learning a heuristic as a regression problem, we aim to minimise the mean squared error (MSE) loss function during training. The MSE calculates the squared difference between the optimal heuristic value and the heuristic estimate computed by the STRIPS-HGN.

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 in each time step $t \in \{1, \ldots, M\}$ into a heuristic value. We denote the heuristic value output by a STRIPS-HGN after t time steps as h^θ_t. Our training algorithm averages the losses of the 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]. Now, the MSE loss function is defined as:

$$\mathcal{L}_\theta(\mathcal{B}) = \frac{1}{|\mathcal{B}|} \sum_{(G,\, h^*(G)) \in \mathcal{B}} \frac{1}{M} \sum_{t \in \{1,\ldots,M\}} \Big(h^\theta_t(G) - h^*(G) \Big)^2 \tag{4.3}$$

where $\mathcal{B} \subseteq \mathcal{T}$ is a *minibatch* within the entire training dataset $\mathcal{T}$, M is the number of processing steps for the core HGN block, G is the input HGN hypergraph, and $h^*(G)$ is the optimal heuristic value for the state represented by the hypergraph G.

At each optimisation step for a minibatch within an *epoch*, we use the gradient of the MSE loss function $\frac{d\mathcal{L}_\theta(\mathcal{B})}{d\theta}$ to update the weights θ in the direction that minimises the L_θ using an optimiser such as Adam [Kingma and Ba, 2014]. This weight update strategy is called *minibatch stochastic gradient descent*, and is applied to all the minibatches of $\mathcal{T}$ for a given epoch.

Recall than an epoch refers to a single pass over all the samples in the training dataset $\mathcal{T}$. We repeatedly update the weights until we reach a maximum number of epochs MAX-EPOCHS, or until the maximum training time is exceeded.

Finally, after we have finished training a STRIPS-HGN, we can compute a heuristic value for a new problem P and state s by passing the hypergraph generated by

GenerateHGNHypergraph (see Algorithm 4) to the network. We use the output of the decoding block after the last message passing step at time step $t = M$, as the heuristic value.

Minibatch Stochastic Gradient Descent (SGD)

Minibatch SGD updates the weights of a STRIPS-HGN based on the gradients computed from a small subset of the entire dataset $\mathcal{B} \subseteq \mathcal{T}$, which is called a minibatch [Li et al., 2014]. This stands in contrast to batch gradient descent, which updates the weights based on the gradients computed over the entire dataset $\mathcal{T}$ (i.e., a batch size of $|\mathcal{T}|$). Computing the gradient over the entire dataset is not only computationally expensive, but also results in the slow convergence of the loss function.

On the other hand, vanilla stochastic gradient descent (SGD) updates the weights based on the gradient computed over a single sample in $\mathcal{T}$ (i.e., a batch size of 1). Although plain SGD allows the loss function to converge very quickly, the updated weights can contain a substantial amount of noise if proper regularisation techniques are not applied.

In comparison to batch gradient descent and vanilla SGD, Minibatch SGD is designed to provide a good trade off between convergence speed and minimising noise in the trained model. Despite this, our experiments found that a batch size of 1 resulted in a trained STRIPS-HGN with the best planning performance, for reasons that will be explained in Section 5.1.

Chapter 5

Empirical Evaluation

In Chapter 4, we presented the architecture of STRIPS-HGNs, our Hypergraph Network for learning heuristics over the hypergraph induced by the delete relaxation of a planning problem. Moreover, we discussed our training data generation procedure, and our algorithm for optimising the weights of the update functions within a STRIPS-HGN.

Now, we will train our STRIPS-HGNs on a variety of planning domains to evaluate their efficacy in comparison to the baseline heuristics h^{add}, h^{max} and LM-cut. Our results demonstrate that the heuristics we learn can significantly reduce both the number of heuristic calls and search time, whilst maintaining near-optimal solutions.

Section 5.1 will discuss our experimental setup, including the spatial-based and spectral-based HGN configurations we use, the *invariance* of these networks, and our stratified k-fold training procedure which aims to restrict noise and demonstrate robustness over the generated training sets. Section 5.2 will then examine the domains we train and evaluate our HGNs on, while subsequent sections will present and analyse the results of our experiments.

5.1 Experimental Setup

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. Unless otherwise specified, we **repeat each experiment 10 times**, resulting in 10 different trained networks. This is done to minimise the influence of the randomly generated problems and the training procedure.

The rest of this section discusses the search and STRIPS-HGN configurations, our training procedure, and how the plots of our results may be interpreted.

Hardware

All experiments were conducted on a Amazon Web Services `c5.2xlarge` server with an Intel Xeon Platinum 8000 series processor with 4 physical cores (8 logical cores), and 16GB of RAM. Each experiment was limited to a single physical core with a turbo clock frequency of 3.4Ghz. We did not set a limitation on the amount of memory an experiment used, however, as we observed each experiment never exceeded 2GB.

5.1.1 Search Configuration

We evaluate the heuristics learned by our HGNs on A* search as they were trained on the optimal heuristic values. Although it is possible to use Greedy Best-First Search (GBFS) which only considers the ranking of states [Pearl, 1984], we believe that the heuristic estimates are still informative for A* as we do not expect them to deviate significantly from the optimal values.

To generate the training data for each training problem, we used Fast Downward (FD) [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. Pyperplan is a Python-based classical planner, while FD is based on C++.

We used Pyperplan to evaluate all the heuristics as STRIPS-HGNs are implemented in Python, and for the ease of collecting detailed metrics to compare the performance of each heuristic. We observed that the implementations of the delete-relaxation heuristics in Pyperplan are much slower than their counterparts in FD. This suggests that we should move our evaluation to FD as future work.

We compare the performance of A* with the learned heuristics against the performance of A* with h^{add}, h^{max} and LM-cut.

5.1.2 Hypergraph Network Configuration

We generate the hypergraph of each planning problem by using the delete-relaxed problem computed by Pyperplan. The full hypergraph generation procedure is detailed in Section 4.1.

The expected input to a spatial-based STRIPS-HGN is the hypergraph representation of the delete-relaxed planning problem, along with the proposition (vertex) features and action (hyperedge) features for the current state. For our spectral-based HGNN approach, the input is the normalised incidence matrix of the hypergraph and the proposition features only, as a HGNN does not support hyperedge-level input features.

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 $p \in F$ as a vector $[x_s, x_g]$ of length 2 where:

$$x_s = \begin{cases} 1, & \text{if } p \in s \\ 0, & \text{otherwise} \end{cases} \qquad \text{and} \qquad x_g = \begin{cases} 1, & \text{if } p \in G \\ 0, & \text{otherwise} \end{cases}$$

Recall that G is the set of goal propositions. Observe that x_s indicates whether a proposition p is in the current state, while x_g indicates whether p is a goal proposition. Consequently, propositions that are neither in the current state nor in the goal state have the feature $[0, 0]$. We may stack the features for each proposition to get a matrix $\mathbf{X} \in N^v \times 2$, where $N^v = |F|$ is the total number of propositions for the problem.

Spectral-based HGNN

For our spectral approach, we use a Hypergraph Neural Network with two hyperedge convolutional layers (HGNN_1 and HGNN_2) with increasing output dimensionalities, followed by two fully connected layers (FC_1 and FC_2) which aggregate the hidden representations of each proposition back to 1 dimension. We apply the LeakyReLU activation function [Maas et al., 2013] after each hidden layer.

Recall that a HGNN only accepts vertex-level inputs and returns vertex-level outputs. Hence, to get the final heuristic value, we sum the 1-dimensional vertex-level outputs of the final layer FC_2 together. This architecture is depicted in Figure 5.1. Although it is possible to adapt STRIPS-HGNs to represent the architecture of our spectral-based HGNN, we do not do so for simplicity's sake.

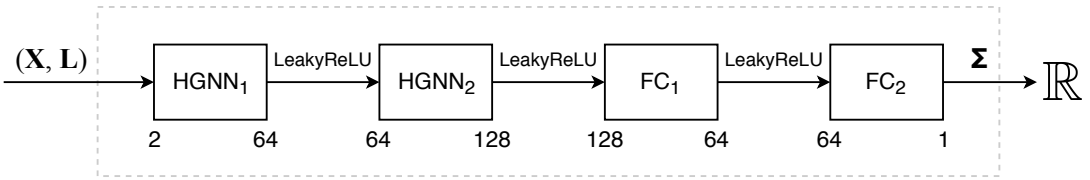


Figure 5.1: The architecture of our spectral-based network for learning heuristics. The numbers below each layer represent its input vertex dimensionality and output vertex dimensionality. $\mathbf{L}$ is the normalised incidence matrix which is only used by the HGNN layers.

The input to our HGNN is a tuple $(\mathbf{X}, \mathbf{L})$, where $\mathbf{X} \in N^v \times 2$ are the stacked vertex (proposition) features for the current state, and $\mathbf{L} = \mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}\mathbf{D}_e^{-1}\mathbf{H}^T\mathbf{D}_v^{-1/2}$ is the normalised incidence matrix of the hypergraph for the delete-relaxed planning problem. Since the hypergraph structure for a delete-relaxed problem is fixed, we only need to compute $\mathbf{L}$ once for a given problem.

Additionally, as a HGNN only supports undirected hypergraphs, we must convert the directed hypergraph induced by the delete-relaxed STRIPS problem into an undirected hypergraph. We do this by simply taking the absolute value of each element in the incidence matrix of the directed hypergraph (see Section 3.1 for definitions).

The intuition behind our HGNN architecture is that the network will learn how much each proposition contributes to the optimal heuristic value by using the normalised incidence matrix to pass signals from each vertex to its neighbouring vertices. This will ideally enable the network to generalise to new states it was not trained on. As we will show in our experimental results, the generalisation capability of spectral-based HGNs is limited in comparison to that of spatial-based HGNs. For this reason, the main focus of our experiments is to demonstrate the effectiveness of the latter. We denote the heuristic learned by our spectral approach as $h^{spectral}$.

Spatial-based STRIPS-HGNs

Our spatial-based HGN uses the STRIPS-HGN architecture introduced in Section 4.2.2. The input feature for each vertex (proposition) is the vector $[x_s, x_g]$ defined previously. The input feature for each action a represented by a hyperedge e is a vector $[w_e, r_e, s_e]$, where w_e is the cost of the action a, and r_e and s_e are the number of positive effects and preconditions for action a, respectively. r_e and s_e are used by a STRIPS-HGN to determine the number of receivers and senders a hyperedge has. The maximum number of receivers $N_{receiver}$ and senders N_{sender} for a STRIPS-HGN is fixed a priori – this is described in the details for the Core Block below.

We set the number of message passing steps M for the recurrent core HGN block to 10, and let the global (i.e., encoded heuristic value), vertex, and hyperedge features each have a hidden dimensionality of 32. The update functions of all the HGN blocks in a STRIPS-HGN are implemented as MLPs which share identical architectures. Each MLP consists of two fully-connected layers (FC_1 and FC_2), with each layer being followed by a LeakyReLU activation function. This design is depicted in Figure 5.2.

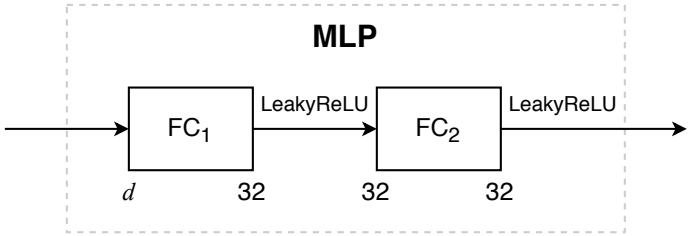


Figure 5.2: The MLP architecture used by all the update functions in a STRIPS-HGN. The numbers below each layer represent its input dimensionality and output dimensionality; the dimensionality of the input to the MLP is d. There are two fully connected layers, each followed by the LeakyReLU activation function.

In the MLP for the update function ϕ^u for the decoding block, we add an extra fully connected layer which transforms the latent representation of the global attributes (32 dimensions) into a single heuristic value. The MLPs are initialised with weights generated using the Kaiming procedure with a uniform distribution [He et al., 2015], which helps in avoiding exploding or vanishing gradients during training.

We refer the reader back to Section 4.2.2 for details regarding the input features to each update function in the encoding, core, and decoding blocks of a STRIPS-HGN. We concatenate each update function's input features before they are passed to the MLP. Table 5.1 summarises the fixed input dimensionalities for the MLP update functions within each block, which we describe in detail below:

- **Encoding Block**: the input dimensionality of the vertex-level MLP is 2, as the feature for each proposition is a vector $[x_s, x_g]$ of length 2. The input dimensionality of the hyperedge-level MLP is 3, as the feature for each hyperedge is a vector $[w_e, r_e, s_e]$ of length 3.
- **Core Block**: recall that the input hypergraph to the core block (see Section 4.2.2) is the concatenation of the initial encoding of the hypergraph G^0_{hid} and the

hypergraph previously output by the core block G_{hid}^{t-1}. Hence, the vertex and hyperedge features in the concatenated input hypergraph each have dimensionality of $2 \times 32 = 64$.

- Hyperedge-level MLP: the input to the hyperedge-level update function are the features $\mathbf{e} \in \mathbb{R}^{2\times 32} = \mathbb{R}^{64}$ for a hyperedge $(\mathbf{e}, R, S)$ in the input hypergraph; along with the concatenated features for the receiver vertices and sender vertices for the hyperedge which have dimensionalities $2 \times 32 \times |R| = 64|R|$ and $2 \times 32 \times |S| = 64|S|$, respectively.

  However, this raises an issue – the input dimensionality of the MLP must be fixed, but the number of receiver and sender vertices may vary across actions. We can compute the maximum number of receivers $N_{receiver}$ (senders N_{sender}) by taking a maximum over the number of positive effects (preconditions) for each action schema in the domain definitions for several domains[1]. Hence, to satisfy the fixed input dimensionality requirement of a MLP, we use $N_{receiver}$ and N_{sender} to fix the maximum dimensionality required by the concatenated receiver and concatenated sender vertex features, as depicted in Figure 5.3. We sort the receiver vertices and sender vertices alphabetically by their corresponding proposition names, and insert their features in **sorted order** into the concatenated feature vectors of dimensionality $64 \times N_{receiver}$ and $64 \times N_{sender}$, respectively. For actions with fewer receiver and sender vertices than the maximums, we pad the remaining space in the feature vector with zeros.

  To sum up, we concatenate the hyperedge features, receiver vertex features and sender vertex features (both padded with zeros if necessary) to get an input dimensionality of $64 + 64|N_{receiver}| + 64|N_{sender}| = 64(1 + N_{receiver} + N_{sender})$. An example of this complete concatenated input feature is presented in Figure 5.3.
- Vertex-level MLP: the input to the vertex-level update function are the features $\mathbf{v} \in \mathbb{R}^{2\times 32} = \mathbb{R}^{64}$ for a vertex in the input hypergraph, along with the aggregated latent representation of the hyperedges where the vertex is a receiver which has a dimensionality of 32. We concatenate these features to get an input dimensionality of $64 + 32 = 96$ for the vertex-level MLP.
- Global MLP: the input to the global update function is the aggregated latent representation of both the hyperedges and the vertices, each having a hidden dimensionality of 32. We concatenate these features to get an input dimensionality of $32 + 32 = 64$.

- **Decoding Block**: the input dimensionality of the global MLP in the decoder is the hidden dimensionality of the global attributes, which is 32.

For the aggregation functions $\rho^{e\to v}$, $\rho^{e\to u}$ and $\rho^{v\to u}$ in the core HGN block of a STRIPS-HGN, we use element-wise summation. This satisfies the permutation

[1]This approach would not work for the universally quantified preconditions and effects found in the ADL fragment of PDDL. However, it works fine for STRIPS.

	Encoding Block	Core Block	Decoding Block
Global MLP	$\emptyset$	$32 + 32 = 64$	32
Vertex MLP	2	$2 \times 32 + 32 = 96$	$\emptyset$
Hyperedge MLP	3	$2 \times 32(1 + N_{receiver} + N_{sender})$	$\emptyset$

Table 5.1: Input dimensionalities for the MLPs in the Encoding, Core, and Decoding HGN blocks of a STRIPS-HGN. $N_{receiver}$ and N_{sender} are the maximum number of receivers and senders for all hyperedges, respectively. $\emptyset$ indicates that the block does not contain a MLP for the corresponding update function.

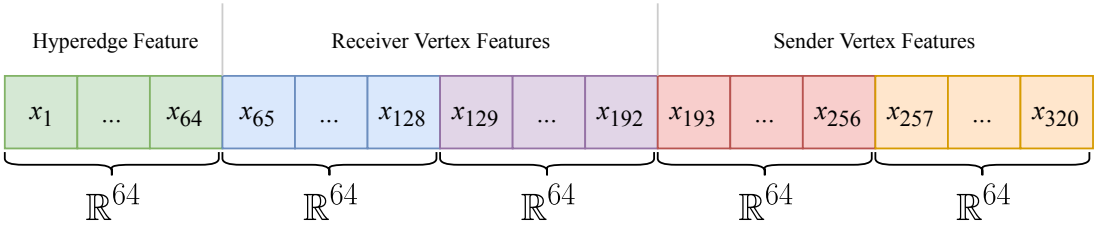


Figure 5.3: The concatenated features for a single hyperedge which is the input to the Hyperedge-level MLP in the core processing block of a STRIPS-HGN. We assume $N_{sender} = N_{receiver} = 2$. If a hyperedge contains only a single receiver vertex, we insert the latent feature for that vertex into the space between x_{65} and x_{128}, and set all elements between x_{129} and x_{192} to 0 (zero padding). On the other hand, if a hyperedge contains two receiver vertices, we firstly alphabetically sort the latent vertex features by their proposition names. We then insert the features of the first vertex between x_{65} and x_{128}, and the features of the second vertex between x_{129} and x_{192}.

invariant and variable number input requirements of an aggregation function in a HGN block.

As mentioned in Section 4.2.3, the intuition behind our spatial STRIPS-HGN design is that we incrementally propagate the latent features for the vertices and hyperedges from the initial propositions to the goal propositions. For problems where the optimal plan requires more than 10 actions, 10 message-passing steps is insufficient for the information from the initial propositions to reach **all** the goal propositions. Nevertheless, our results show that the heuristic estimates provided by our STRIPS-HGNs help A* find a near-optimal plan in less heuristic calls than h^{max} and LM-cut even when the optimal plan has more than 10 actions. We denote the heuristic learned by our spatial approach as $h^{spatial}$.

Implementation Details

The design of our implementation of Hypergraph Networks is based off the implementation of Graph Networks: https://github.com/deepmind/graph_nets/. In contrast to the GNs framework, which is implemented in Tensorflow and Sonnet, our HGNs framework is implemented in PyTorch. We implement STRIPS-HGNs using our HGN framework.

Since HGNs and GNs are designed to be extremely general, the resulting models

that may be implemented are not very fast due to the overheads of the framework. Models which do not require the full representational capability of HGNs should be implemented from the ground up to maximise their computational efficiency. Moreover, there are several optimisations that can be made to increase the training and evaluation speed of our HGN framework, including improving implementation details and using multiple CPU cores or even a GPU in our experiments. Hence, the CPU times in our results for $h^{spatial}$ should be considered preliminary.

The spectral-based HGNN model was implemented using the reference source code provided by Feng et al. [2019]: https://github.com/iMoonLab/HGNN. We did not use our HGN framework to implement HGNNs, as the reference implementation is extremely fast.

Invariance of HGNNs and STRIPS-HGNs

We define the invariance of a HGN as the property where the network returns identical heuristic estimates for *isomorphic* states in a set of *isomorphic* problems. We call two problems P_1 and P_2 isomorphic if we can rename the objects, predicates (propositions), and actions of P_2 to get P_1. Two states s_1 from P_1, and s_2 from P_2 are isomorphic if the renaming of P_2 such that $P_2 = P_1$ guarantees $s_2 = s_1$. Consider the isomorphic Blocksworld problems shown in Figure 5.4 – we would expect a learned heuristic to return identical heuristic estimates for all *isomorphic* states in the problems. Similarly, we call two hypergraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ isomorphic[2] if there exists a relabelling of the vertices and hyperedges in $\mathcal{G}_2$ such that $\mathcal{G}_2$ is identical to $\mathcal{G}_1$.

A spectral-based HGNN computes a new feature for each vertex v by performing a weighted aggregation of the features of its neighbouring vertices, where the normalised incidence matrix is used to define these neighbours. Since a HGNN computes feature updates based on a hypergraph's underlying structure, it is invariant to different orderings of the vertices and hyperedges in the incidence matrix, as long the underlying hypergraphs are isomorphic. Consequently, a HGNN is invariant to the renaming of objects in a planning problem, and the renaming of actions and predicates in the domain definition.

On the other hand, a STRIPS-HGN is only invariant to the renaming of objects in a problem and the renaming of actions in the domain definition. The lack of invariance to the renaming of predicates in the domain definition may be attributed to the alphabetical sorting procedure we impose on the receiver and sender vertices before we concatenate them as input to the hyperedge-level MLP in the update function ϕ^e in the core block. Although we could define a more intelligent sorting procedure which maintains a stricter form of consistency, in order to guarantee invariance the update function ϕ^e must be implemented as a model, such as Deep Sets [Zaheer et al., 2017], which is permutation invariant to the ordering of the receiver and sender vertices. Although implementing this is not trivial, it is feasible given the flexibility of the HGN framework. Despite this invariance limitation, the generalisation capability

[2] Although this is not the formal definition of a hypergraph isomorphism, it suffices for our discussion. We refer the reader to [Luks, 1999] for the formal definition.

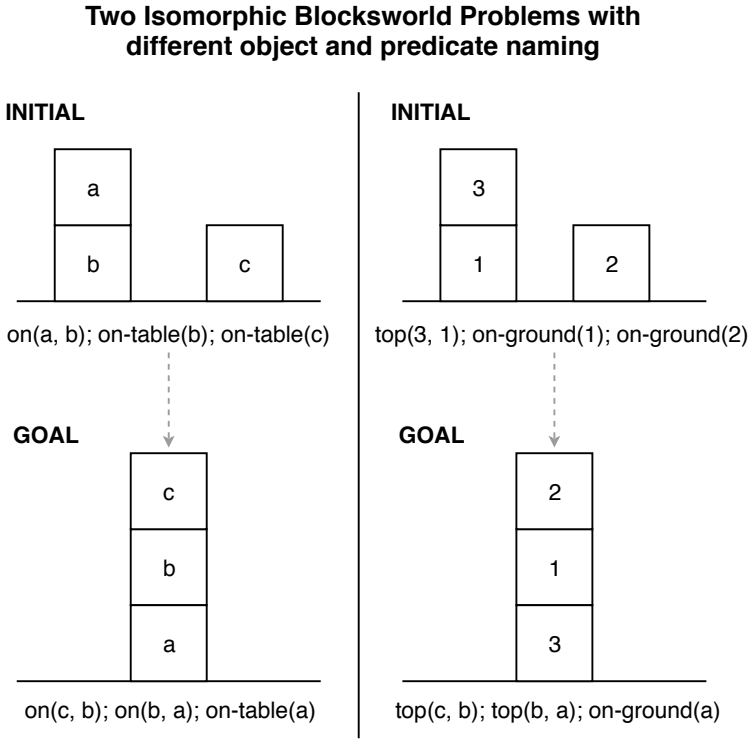


Figure 5.4: Two isomorphic Blocksworld problems where the blocks and predicates have different names.

of a STRIPS-HGN is extremely promising.

5.1.3 Training Procedure

Stratified k-Fold and Binning

k-Fold is a technique to split a dataset into k folds (i.e., partitions), with each fold containing roughly the same number of samples. Stratified k-Fold is an extension to k-Fold which guarantees that each fold approximately maintains the percentage of samples for each target class found in the dataset. This increases the probability that the fold is representative of the entire dataset.

Stratified k-Fold requires the target heuristic values to be discrete, not continuous. We use numerical binning to split the continuous values into n discrete bins – this converts the continuous values into discrete values. We are only concerned with quantile binning, where the dataset is split into n bins of approximately equal size by using the percentiles of the continuous input values (e.g. for $n = 4$ bins, we use the 25th, 50th and 75th percentiles). We refer the reader to https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.KBinsDiscretizer.html for more details regarding the quantile binning procedure.

In our training procedure, we firstly bin the samples in our training dataset based

on their target optimal heuristic values. We then use stratified k-Fold to split the dataset into k folds $F = \{f_1, \ldots, f_{10}\}$ such that each fold $f \in F$ contains roughly the same number of samples for each heuristic bin, ensuring that the fold is relatively representative of the dataset. Now, we may train a separate HGN for each fold $f \in F$ using $F \setminus f$ as the training set and f as the validation set which we use to detect overfitting. Evidently, this will mean that every training sample is used for both training and validating a model (but **not** the same model). Recall that the validation set is used to test a network's ability to generalise to data it was not trained on.

k-Fold is usually used for cross validation, where the effectiveness of different choices of model design is compared by analysing the mean validation loss across the k folds for each model design. However, we use k-Fold to reduce any potential noise and demonstrate robustness over the training set used, by selecting the single model which achieved the lowest loss for the validation set in its fold as the 'representative' for an experiment.

In planning, we may generate as much training data as we want by using a problem generator, which is equivalent to a generative model of the domain. This is in comparison to a typical machine learning dataset, where the samples have to be painstakingly collected and tagged either manually or semi-automatically. Hence, we use k-Fold to show that our Hypergraph Networks are robust with respect to the problems generated for training, and to reduce any noise that may arise from an unrepresentative training/validation split of the dataset which may lead a HGN to overfit/underfit to the true distribution of the dataset.

Training Configuration

Unless otherwise specified, the same configurations which we describe below are maintained across each experiment. After generating the training data using the procedure described in Algorithm 4, we split the training data into 4 bins based on the optimal heuristic values, and use stratified 10-Fold cross validation to split the training data into 10 folds $F = \{f_1, \ldots, f_{10}\}$ of roughly equal size.

Now, for each fold $f \in F$, we train a Hypergraph Network using $F \setminus f$ as the training set and f as the validation set, and subsequently select the network at the epoch which achieved the lowest loss on the validation set f. This prevents any overfitting to the training set, and mitigates any effects of noise that may arise from influences such as the minibatch size. Since we train one HGN for each of the 10 folds, we are left with 10 separate networks. We select the network which performed best on the validation set for its fold as the **single representative HGN** for an experiment. This single representative HGN is then evaluate on a previously unseen test set.

Both the spectral-based HGNN and spatial-based STRIPS-HGNs are trained using the procedure we defined in Section 4.3. We 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. Moreover, as one of the goals of our experiments is

to show that a HGN is able to generalise when trained on a small set of problems, the size of our training data is usually limited to 50 to 200 samples. Hence, a larger minibatch size would likely trap the network in a local optimum during training, as the losses (and hence gradients) would be averaged across the minibatch (Equation 4.3).

We concede that our training procedure is both time-consuming and not extremely robust, and consequently has room for improvement. Future work should consider how to more reliably train a HGN in order to maintain more consistent performance across several runs of an experiment. Nevertheless, our modified stratified k-Fold training procedure results in much more reliable performance across several experiments in comparison to the performance of networks trained and selected based on a simple train/validation split.

5.1.4 Interpreting the Result Plots

The metrics which we show in our plots are, for each heuristic, the number of heuristic calls and the total CPU search time required by A* to solve the problem, as well as the deviations from the optimal plan length. Since we run an experiment multiple times for $h^{spectral}$ and $h^{spatial}$, we plot the mean and 95% confidence interval for a metric in the runs where a plan was found by A* in the limited search time.

We order the problems in the x-axis of the plots by the *problem difficulty*. Let $\mathcal{P} = [P_1, \ldots, P_n]$ represent the final ordering for n testing problems which at A* was able to solve with at least one heuristic. We now discuss how $\mathcal{P}$ is constructed.

The heuristics are firstly sorted in ascending order of their coverage on the test set (i.e., the number of testing problems it was able to solve with A*). Unless otherwise specified, ties are broken in the following order: h^{max}, $h^{spectral}$, $h^{spatial}$, LM-cut, h^{add}.

Now, for each heuristic in this ordering, we sort the problems $[P_j, \ldots, P_k]$ which A* was able to solve by the number of heuristic calls and append each problem $P_i \in sorted([P_j \ldots, P_k])$ into $\mathcal{P}$ only if $P_i \notin \mathcal{P}$. After we perform this procedure for each heuristic and problem, $|P| = n$ as required. Evidently, the continuities in a curve for a heuristic indicate the problems which could be solved, while discontinuities indicate the problems which could not be solved.

The x-axes in the plots for deviation from the optimal plan length do not include problems in the ordering $\mathcal{P}$ which A* was unable to solve with either h^{max} or LM-cut within the time limit. This is because we can only compute a deviation if we know the true optimal plan length. Although our plots for the deviations from the optimal plan length are quite noisy, we may still conclude that h^{add} generally performs substantially worse in comparison to $h^{spatial}$ and $h^{spectral}$. We attempted to use more sophisticated methods for problem ordering and different types of plots including scatter plots, but found that our current approach is the easiest for conveying the plan length deviations and their corresponding confidence intervals.

Problem-Size Dependent	Domain-Dependent	Domain-Independent
n-puzzle, Sokoban (spectral)	Blocksworld, Matching Blocksworld, Gripper, Hanoi Ferry, Zenotravel, Sokoban (spatial)	Blocksworld + Zenotravel + Gripper, Train on Zenotravel + Gripper and evaluate on Blocksworld

Table 5.2: The classes of heuristics we learn in our experiments, along with the domains we use in these experiments.

5.2 Domains and Problems

Across all the domains we present below, we assume that all actions have unit cost. However, as we have previously discussed, STRIPS-HGN and HGNNs are able to support actions with different costs through the hyperedge weights. All the problems in our experiments were randomly generated and have unique initial states and goal states. Hence, there is minimal to no overlap between the training problems and testing problems.

The main goal of our experiments is to show that a STRIPS-HGN trained on a set of small-sized problems is able to generalise to significantly larger problems. Our experiments for each domain may be broken down into learning three classes of heuristics (shown in Table 5.2):

1. **Problem-size dependent** heuristics: a heuristic learned by a HGN which has been trained and evaluated on problems from the same domain with identical hypergraph structures (i.e., same number of objects, propositions and actions), where we only vary the initial state and the goal.
2. **Domain-dependent** heuristics: a heuristic learned by a HGN which is trained on small problems from a domain, and evaluated on larger problems from the same domain.
3. **Domain-independent** heuristics: a heuristic learned by a HGN which has been trained on problems from several domains, and evaluated on problems from the same or different domains to the ones the network was trained on.

Sections 5.2.1 to 5.2.8 will describe the domains we evaluate our HGNs on, along with the experimental configurations for learning problem-size dependent heuristics and domain-dependent heuristic heuristics. Although we learn problem-size dependent heuristics and domain-dependent heuristics, the complete pipeline for training and evaluating a HGN is domain-independent. Next, Section 5.2.9 will describe the problems and configuration we use to show that the heuristic function learned by a STRIPS-HGN may generalise across multiple domains, i.e., it is domain-independent to a certain extent.

5.2.1 Blocksworld

An agent's objective in Blocksworld is to use its gripper to stack a set of blocks on the table in some specified goal configuration. We consider 4-operation Blocksworld, where an agent may: *stack* one block on top of another, *unstack* one block from another, *put-down* a block it is holding on to the table, and *pick-up* a block from the table. An example of a Blocksworld problem is shown in Figure 5.5.

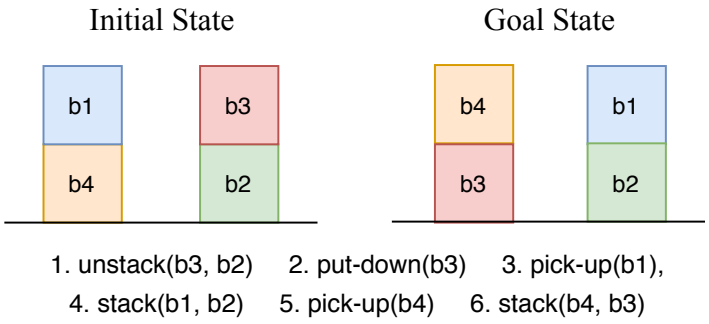


Figure 5.5: Example of a Blocksworld problem with 4 blocks, along with the actions in an optimal plan.

Experimental Configuration

For our Blocksworld experiment, we train both the spectral-based and spatial-based approach to analyse and compare their generalisation capabilities to larger problems. For each of the 10 folds in an experiment, we train the fold's network for 10 minutes. It is important to note that although this gives a total training time of 100 minutes, the final network that we select has only been trained for 10 minutes.

Each network was trained on the optimal training data generated from $10 \times \{3, 4, 5 \text{ blocks}\} = 30$ problems, and evaluated on $20 \times \{6, 7, 8, 9, 10 \text{ blocks}\} = 100$ larger problems. We generate the problems using the `BWSTATES` generator described in [Slaney and Thiébaux, 2001], which is publicly accessible here: http://users.cecs.anu.edu.au/~jks/cgi-bin/bwstates/bwcgi.

5.2.2 Matching Blocksworld

Matching Blocksworld is a variant of Blocksworld where each block has a positive or negative polarity, and the agent now has two grippers, one with positive and one with negative polarity [Fern et al., 2011]. When a gripper picks up a block of the opposite polarity, the block becomes damaged such that no other block may be stacked on top of it. This may evidently lead to a dead end.

The difficulty in a Matching Blocksworld problem is determined by the number of blocks, and the number of towers required in the goal state. As we increase the number of blocks and decrease the number of towers, a problem becomes more difficult.

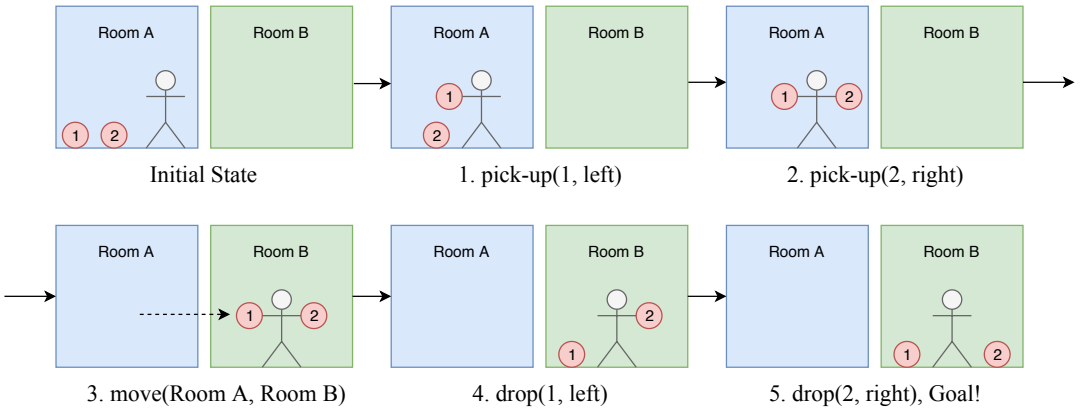


Figure 5.6: Example of a Gripper problem with 2 balls, along with the optimal plan. Note that actions have been abbreviated from their original domain definition for simplicity.

Experimental Configuration

We train our STRIPS-HGNs on $5 \times \{2, 3, 4 \text{ blocks}\} \times \{1, 2 \text{ towers}\} = 30$ problems, and evaluate them on 89 larger problems with 5 to 8 blocks. The testing set consists of 19 problems for 5 blocks (6, 6, 5, and 2 problems for 1, 2, 3, and 4 towers, respectively); 20 problems for 6 blocks ($5 \times \{1, 2, 3, 4 \text{ towers}\}$); and 25 problems each for 7 and 8 blocks ($5 \times \{1, 2, 3, 4, 5 \text{ towers}\}$).

ll problems were randomly created using the generator provided in the IPC 2008 Learning Track competition [Fern et al., 2011]. We limit the training time for the network in each of the 10 folds to 15 minutes, giving a total training time of 2.5 hours (150 minutes).

5.2.3 Gripper

Gripper is a very simple domain where a robot, which has 2 grippers (left and right), must move n balls from Room A to Room B [Long et al., 2000]. A Gripper problem with $n = 2$ balls is depicted in Figure 5.6. The optimal plan for a problem of any size requires the robot to pick up 2 balls (or 1 ball if only one is available) from Room A using its two grippers, move to Room B, drop the balls, move back to Room A, and repeat. Despite the fact that Gripper is a simple problem, A* with h^{max}, h^{add}, and LM-cut struggles as the number of balls increases above 10.

Experimental Configuration

We train both spatial and spectral HGNs on the first three Gripper problems with 1 to 3 balls, and evaluate them on problems with 4 to 20 balls. We reduce the number of bins to 3 given the limited size of the training set, and restrict the training time of each fold's network to 90 seconds. This gives a total training time of 15 minutes for the 10 folds in a given experiment.

Moreover, since we only generate a training set with 20 samples from the training

problems, we resample the training set to size 50 using stratified resampling with replacement (the binned heuristic values are used as the class labels). The heuristic bins are then recomputed based on the resampled training set before stratified 10-fold is applied. Due to the noise in this resampling procedure which may result in networks with varying performance, we repeat the experiment 20 times.

Interestingly, our spatial-based STRIPS-HGN performed worse in terms of planning performance if we added the problem with 4 balls to the training set. This suggests that the network may be overfitting to the training data, and hence a more suitable training mechanism should be deployed.

5.2.4 Hanoi

In the Tower of Hanoi, which we abbreviate to Hanoi, there are three pegs and n disks of different sizes. In the initial state the n disks are stacked on top of each other (by ascending size) on the left-most peg. The goal is to move the n disks to the right-most peg and maintain this ordered stack. The only action we may apply in any state is to move a disk at the top of the stack from one peg to another peg, where a disk cannot be placed on top of a smaller disk. Figure 5.7 depicts a Hanoi instance with $n = 3$ disks.

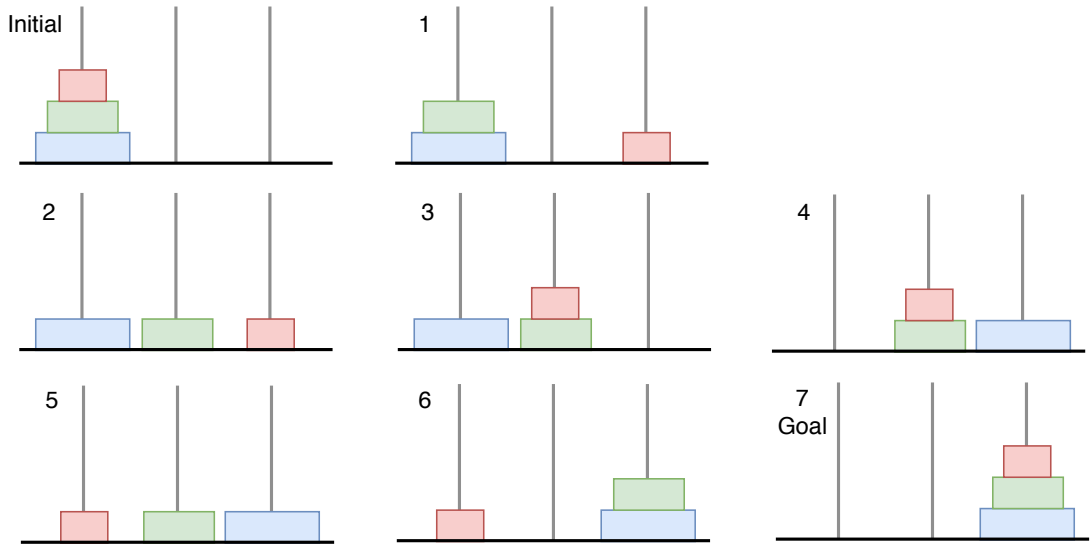


Figure 5.7: Example of a Hanoi problem with 3 disks, along with the seven states in the optimal plan.

The key to Hanoi is to notice that a problem with n disks can be solved recursively by breaking it down into simpler computations for $n-1, n-2, \ldots, 1$ disks [Pierrot et al., 2019]. For the base case with 1 disk, we can move the disk to the right peg without any restrictions. We would expect HGNs to struggle on Hanoi, as learning this recursive relationship is quite difficult for a learning-based model.

Experimental Configuration

We train a STRIPS-HGN on the Hanoi problems with 3 and 4 disks, and evaluate the network on problems with $3, \ldots, 10$ disks. We train the network in each of the 10 folds for 5 minutes, giving a total training time of 50 minutes.

Since the training set only contains 24 training pairs, we resample it to size 50 using stratified resampling with replacement using the initial heuristic bins which are subsequently recomputed (as described for Gripper in Section 5.2.3). The overlap of the training and testing set for problems with 3 and 4 disks intends demonstrate show the poor planning performance of the STRIPS-HGN on problems it was trained on. As we will show, a STRIPS-HGN is unable to learn an informative heuristic value for Hanoi.

5.2.5 Ferry

Ferry is a transportation problem where the objective is to move a number of cars from their initial locations to their goal locations using a ferry. A ferry may travel from one location to any other location, but may only carry one car at a time. We refer the reader to [Long and Fox, 1999], which contains an analysis of not only Ferry, but also Gripper and Hanoi.

Experimental Configuration

We train a STRIPS-HGN on $\{2, 3, 4 \text{ locations}\} \times \{1, 2, 3, 4 \text{ cars}\} = 12$ problems, and evaluate the network on $\{2, 3, \ldots, 10 \text{ locations}\} \times \{5, 10, 15, 20 \text{ cars}\} = 36$ significantly larger problems.

We decrease the number of folds to 5, as the generated training which contains 70 samples is quite small. Additionally, we limit the training time for the network of each fold to 3 minutes. This gives a total training time of 15 minutes for each experiment.

5.2.6 Zenotravel

Zenotravel is a transportation problem where the objective is to move each passenger from their origin city to their desired destination city by using one or more planes. Each plane has fuel which may need to be replenished, and can only carry one passenger at a time. Evidently, Zenotravel is a more complex transportation problem than Ferry.

A Zenotravel problem has a fixed number of passengers, cities, and planes. The difficulty of a problem is heavily influenced by the number of passengers, as it is the factor which exerts the most impact on the depth of the search tree. We refer the reader to [Helmert, 2008] for more details regarding Zenotravel.

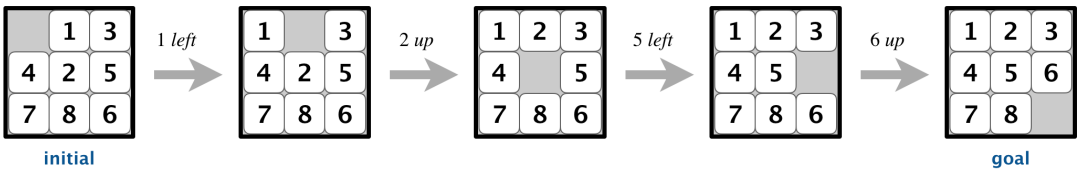


Figure 5.8: Example of an 8-puzzle problem along with its optimal plan. Figure taken from https://www.cs.princeton.edu/courses/archive/spring18/cos226/assignments/8puzzle/index.html

Experimental Configuration

We train a STRIPS-HGN on 10 problems with 2 cities which have 1-4 planes and 2-4 passengers, and 10 problems with 3 cities which have 1-3 planes and 2-4 passengers. We evaluate the trained network on $\{2,3,4 \text{ cities}\} \times \{2,3,4,5 \text{ planes}\} \times \{3,4,5,6,7 \text{ passengers}\} = 60$ problems. We use the generator provided in the IPC 2002 planning competition [Long and Fox, 2003] to randomly generate the problems.

We train the STRIPS-HGN in each fold for 10 minutes, giving a total training time of 100 minutes for each experiment.

5.2.7 n-puzzle

n-puzzle is a sliding puzzle problem where there are n tiles in a square grid with $n+1$ squares. Let us assume the names of the tiles are $1,\ldots,n$. The objective of n-puzzle is to slide the tiles from their initial configuration to the goal configuration where the tiles are ordered alphabetically by their names from left-to-right and top-to-bottom (see the goal state in Figure 5.8).

Although finding a plan to a n-puzzle problem is relatively easy, finding an optimal plan is difficult and has been shown to be NP-hard [Ratner and Warmuth, 1986]. Approaches which learn macro-actions, which are sequences of actions which have been observed frequently in the plans for a given domain, have shown to be successful for n-puzzle [Iba, 1989].

Experimental Configuration

We train both the spectral-based and spatial-based HGNs on 10 randomly generated 8-puzzle problems with a 3×3 grid size, and evaluate them on 50 randomly generated 8-puzzle problems. The heuristic learned by a HGN in our n-puzzle experiment is problem-size dependent, as the hypergraph structure is identical for the training and testing problems.

We used the generator provided by the IPC 2008 Learning Track competition [Fern et al., 2011] to randomly generate the problems. The goal state for all the training and testing problems are identical, but the initial states are all unique.

We limit the training time for the network in each of the 10 folds to 10 minutes, giving a total training time of 100 minutes for each experiment. We did not experiment with 15-puzzle problems with a 4×4 grid size, as our initial experiments found

that the majority of these problems are too difficult for h^{max} and LM-cut to solve optimally within the limited search time.

5.2.8 Sokoban

An agent's objective in Sokoban is to move boxes in a warehouse from their initial positions to the desired storage locations. We assume that a warehouse is a $n \times n$ grid where each square is either a wall, or the floor. The agent can move and push boxes around on the warehouse floor in the cardinal directions (up, down, left, right), as long as a wall does not block the agent or the box. Figure 5.9 depicts an example of a Sokoban problem.

Sokoban is PSPACE-complete [Culberson C., 1997] and is extremely difficult for search algorithms due to its large branching factor and the substantial search depth required to find a solution. Consequently, this means that it is more difficult for a network to learn a heuristic for Sokoban that it is to learn a heuristic for a problem that is only NP-hard, such as optimal Blocksworld planning.

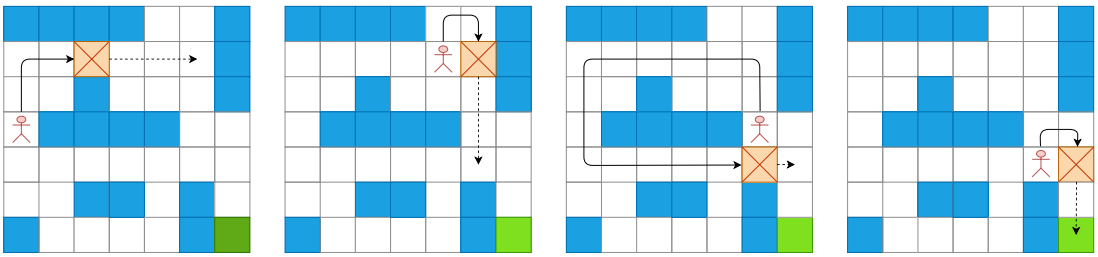


Figure 5.9: Example of a Sokoban problem with 1 box and a grid size of 7. The steps for an optimal plan are indicated in each diagram, where the solid arrows depict the agent moving while the dotted arrow represents the agent pushing the box. The walls are indicated in blue, the box in orange, and the goal position in green.

Experimental Configuration

We limited the number of boxes to 2 for our Sokoban experiments. We generate $10 \times \{5, 7 \text{ grid size}\} = 20$ training problems, and $20 \times \{5, 7 \text{ grid size}\} \times +10 \times \{8 \text{ grid size}\} = 40 + 10 = 50$ testing problems. In each problem, we randomly selected the number of walls to be between 3 and 5. The goal states for problems of a given grid size are identical, due to the implementation of the problem generator provided by [Fern et al., 2011] in the IPC 2008 Learning Track competition.

We increase the number of bins to 5 due to the larger range of optimal heuristic values for the training problems, and reduce the number of folds to 5. The training time for a STRIPS-HGN within each fold is limited to 20 minutes, giving a total training time of 100 minutes for each experiment.

For our spectral-based approach, we train a separate HGNN for each grid size in $\{5, 7, 8\}$. We do so because the hypergraph structures varies significantly as we alter

the grid size, and hence the generalisation of a HGNN to problems larger than those it was trained on would be significantly limited. Additionally, we generate 5 training problems of grid size 8 in order to evaluate the HGNN on the problems of grid size 8. We limit the training time for a HGNN with grid size 5 to 10 minutes per fold, grid size 7 to 15 minutes per fold, and grid size 8 to 20 minutes per fold. These times were determined by observing the rate of convergence of a HGNN on its training problems.

Hence, we learn learn problem-size dependent heuristics for spectral-based HGNNs, and domain-dependent heuristics for spatial-based STRIPS-HGNs.

5.2.9 Multi-Domain Experiments

We now discuss the experimental configurations of our multi-domain experiments, which aim to show that the heuristics learned by STRIPS-HGNs are to a certain degree, domain-independent.

When the training problems come from multiple domains $\mathcal{D} = \{D_1, \ldots, D_n\}$, we run our binning and stratified k-Fold data splitting procedure separately for the problems in each domain $D \in \mathcal{D}$. This results in a total of kn folds for the n domains. We merge together the k folds by merging the 1st fold for D_1 with the 1st folds for $\mathcal{D} - D_1$, and so on for $D_2, \ldots, D_n$. This results in k merged folds, as depicted in Figure 5.10. This procedure ensures that each fold is relatively representative of the training data within each individual domain, and to remove any cross-domain noise if we performed binning on the training samples from multiple domains.

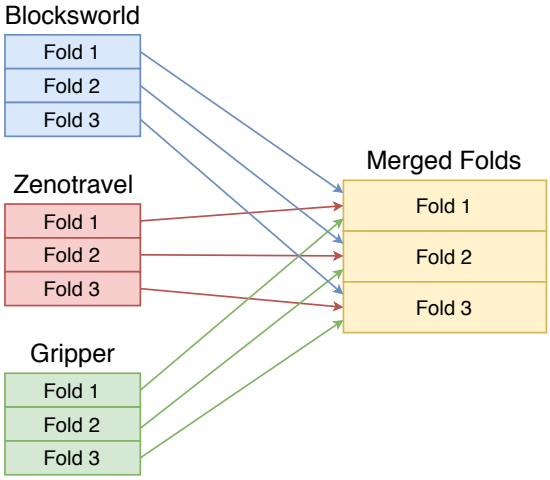


Figure 5.10: The aggregation of 3-folds from 3 separate domains into 3 merged folds.

Training and evaluating on the same domains

We train a STRIPS-HGN on $5 \times \{4, 5$ blocks $= 10$ Blocksworld problems, $5 \times \{2, 3$ cities$\}$ $= 10$ Zenotravel problems (1-4 planes, 2-5 people), and the first 3 Gripper problems

with $\{1, 2, 3$ balls$\}$. The choice of these domains stems from the fact that STRIPS-HGNs are able to provide very informative heuristic estimates when trained individually on each domain. For Gripper, we resample the training set size of 24 to 50 using the stratified resampling procedure previously described in Section 5.2.3.

We limit the training time of the network in each of the 10 folds to 15 minutes giving a total training time of 150 minutes, and evaluate the representative STRIPS-HGN on $20 \times \{6, 7, 8, 9, 10$ blocks$\} = 100$ Blocksworld problems, $\{2, 3, 4$ cities$\} \times \{2, 3, 4, 5$ planes$\} \times \{3, 4, 5, 6, 7$ passengers$\} = 60$ Zenotravel problems, and $\{4, 5, \ldots,$ 20 balls$\} = 17$ Gripper problems. As we will show in our results in Section 5.3.3, a STRIPS-HGNs is able to learn some form of knowledge which allows it to generalise to larger problems across all three of these domains.

Evaluating on different domains to what a STRIPS-HGN was trained on

This experiment aims to show the potential generalisation capability of a STRIPS-HGN to domains it was not trained on. We train a STRIPS-HGN on $5 \times \{2, 3$ cities$\}$ = 10 Zenotravel problems (1-4 planes, 2-5 people), and the first 3 Gripper problems with $\{1, 2, 3$ balls$\}$. For Gripper, we resample the training set size of 24 to 50 using the stratified resampling procedure previously described in Section 5.2.3.

We limit the training time of the network in each of the ten folds to 10 minutes giving a total training time of 100 minutes, and evaluate the learned heuristic on $10 \times \{4, 5, 6, 7, 8\} = 50$ Blocksworld problems.

5.3 Experimental Results

We split our experimental results into three sections based on the classes of heuristics we learn: problem-size dependent, domain-dependent, and domain-independent heuristics. Table 5.2 shows a summary of the domains we experimented with for each class of heuristic. A discussion of our results is provided in Section 5.3.4.

We define the *average coverage* for a heuristic as the average of the coverage across all the test problems. This provides a metric which allows us to roughly gauge how many problems across the test set the heuristic was able to solve. We run A* with h^{max}, h^{add} and LM-cut once giving a coverage of either $0/1 = 0$ or $1/1 = 1$ for each problem. Recall that for $h^{spatial}$ and $h^{spectral}$, we repeat an experiment several times to obtain the test results of multiple trained heuristics (we repeated an experiment 10 times unless otherwise specified in the experimental configurations in Section 5.2). Now, the coverage of a problem is the ratio of the number of STRIPS-HGNs which successfully solved the problem, to the total number of times we repeated the experiment which is equal to the number of STRIPS-HGNs we trained. For example, if we repeated an experiment 20 times but only 8 out of 20 of the STRIPS-HGNs reached the goal when used with A* for a problem P, then P's coverage is $8/20$ for $h^{spatial}$. We report average coverage to 2 decimal places.

5.3.1 Learning Problem-Size Dependent Heuristics

n-puzzle

Figure 5.11 shows the results of our experiments for n-puzzle with 8 tiles, where we trained both the spectral and spatial-based HGNs. All heuristics achieved full coverage for all the test problems.

Firstly, we may notice that h^{max} requires significantly more heuristic calls than all the other heuristics. $h^{spatial}$ and $h^{spectral}$ on average require less heuristic calls than LM-cut and h^{add}. This suggests that the networks have learned some information about the tile configurations within 8-puzzle that helps it provide accurate heuristic estimates.

$h^{spatial}$ requires less heuristic calls than $h^{spectral}$ and maintains a smaller deviation from the optimal plan, but is not competitive in terms of CPU time. This is due to our sub-optimal implementation of Hypergraph Networks and the unavoidable cost of multiple rounds of message passing. On the other hand, the performance of $h^{spectral}$ is very competitive as it requires very little CPU time and finds near-optimal plans.

We observe that both $h^{spatial}$ and $h^{spectral}$ result in significantly lower-cost plans than the plans from h^{add}. $h^{spatial}$ achieves slightly better performance than $h^{spectral}$ in terms of plan length and number of heuristic calls – this may be attributed to the richer message passing design of a STRIPS-HGN which propagates information more effectively in the hypergraph.

Sokoban

Recall from Section 5.2.8 that for each Sokoban experiment, we train a **single** STRIPS-HGN on problems with a grid size of 5 and 7, and evaluate the network on all the testing problems with grid sizes 5, 7 and 8 – this is equivalent to learning a domain-dependent heuristic. In contrast, here in this experiment, we train and evaluate a separate HGNN for each grid size in $\{5, 7, 8\}$ – this means each HGNN learns a problem-size dependent heuristic. We group the results of these heuristics together rather than split them into their respective subsections, so we can easily compare their planning performance.

Figure 5.12 shows the results of our Sokoban experiment. Although the number of heuristic calls for A* with h^{add} is the lowest among all the heuristics, its deviation from the optimal plan length is substantially larger than the deviations for $h^{spectral}$ and $h^{spatial}$.

In general, $h^{spectral}$ requires more heuristic calls than $h^{spatial}$, and has a slightly increased deviation from the optimal plan length. This is despite the fact that we train a separate HGNN for each grid size, providing further evidence for the weaker generalisation capability of spectral-based approaches. Both $h^{spectral}$ and $h^{spatial}$ are competitive with LM-cut in terms of number of heuristic calls. However, $h^{spectral}$ is much more competitive than LM-cut and $h^{spatial}$ in terms of CPU time, given the lower computational cost required by $h^{spectral}$ to compute a heuristic estimate.

The average coverage for the test problems was 1 for h^{max}, h^{add} and $h^{spectral}$, 0.96

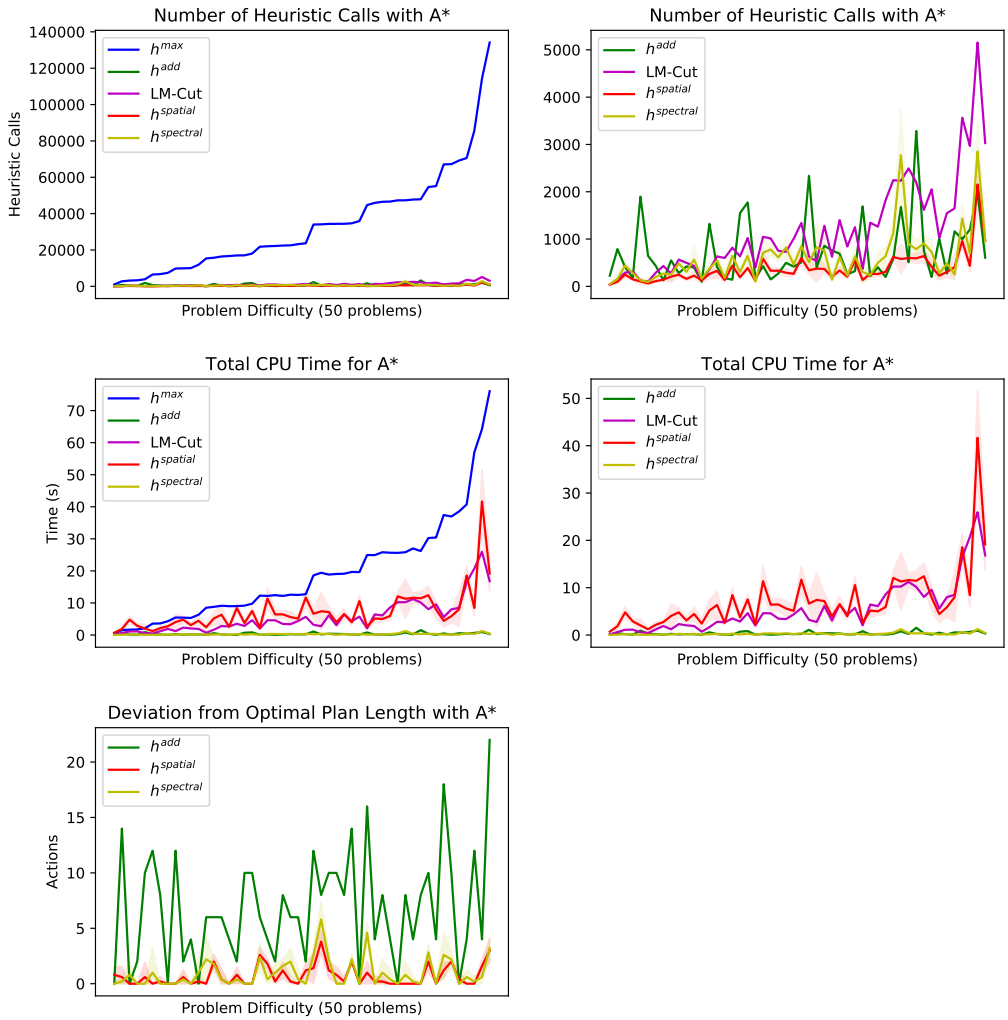


Figure 5.11: Results for n-puzzle with $n = 8$ tiles. h^{max} is removed from the plots in the 2nd column so we can focus on the results for the other heuristics. Note, that the same colour is used for each heuristic across all of our plots.

for LM-Cut, and 0.91 for $h^{spatial}$. The large spike in the confidence interval for $h^{spatial}$ may be attributed to a difficult problem it achieved low coverage for.

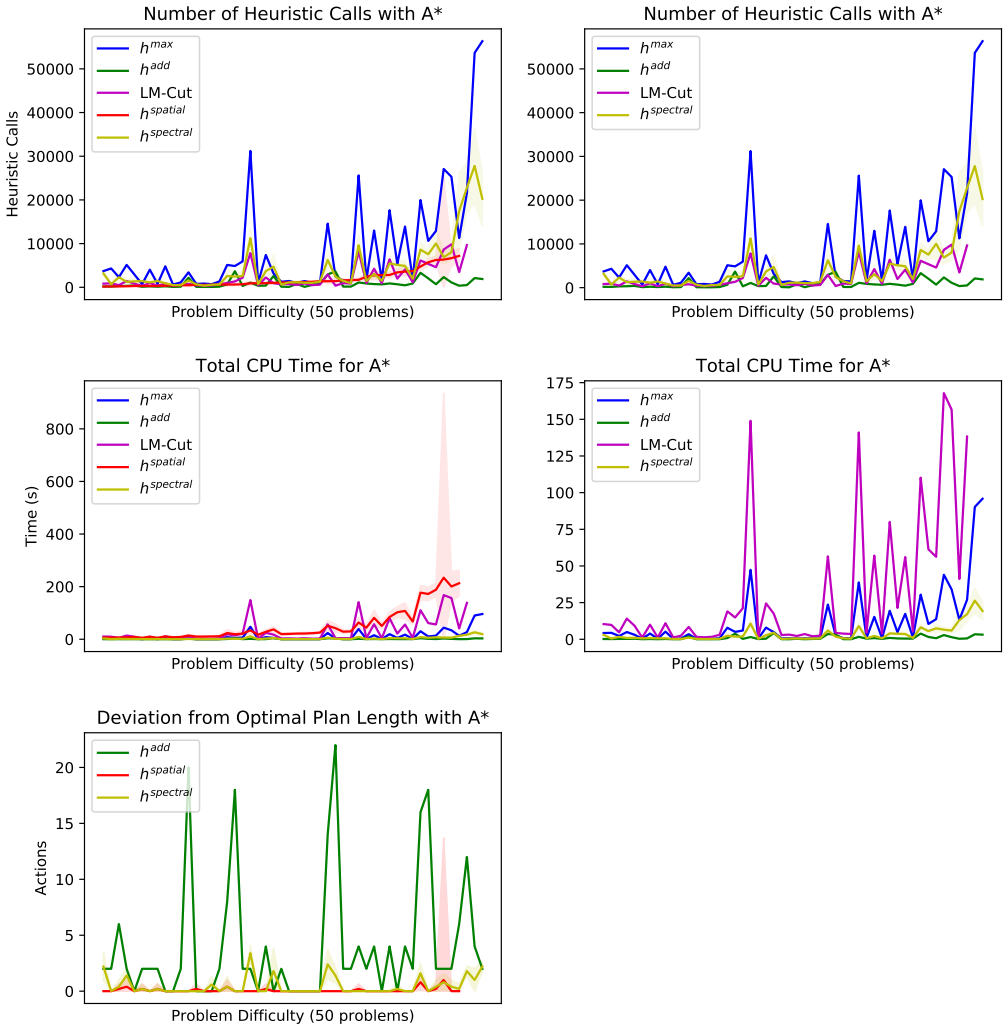


Figure 5.12: Results for Sokoban. $h^{spatial}$ is removed from the plots in the second column. When analysing these plots, it is important to remember that a single STRIPS-HGN is trained on problems with a grid size of 5 and 7, and evaluated on all test problems with a grid size of 5, 7, and 8. In contrast, a separate HGNN is trained and evaluated for each of these grid sizes (see Section 5.9 for details).

5.3.2 Learning Domain-Dependent Heuristics

Blocksworld

Figure 5.13 shows the results of our Blocksworld experiments, where we train both spectral and spatial-based HGNs (see Section 5.2.1). Firstly, we can observe that as the problem difficulty increases, $h^{spectral}$ requires substantially more heuristic calls than $h^{spatial}$. This confirms the limited generalisation capability of spectral-based HGNs. $h^{spatial}$ also requires less CPU time than $h^{spectral}$ for the more difficult problems, and is competitive with LM-cut.

Both $h^{spatial}$ and $h^{spectral}$ deviate significantly less from the optimal plan length in comparison to h^{add}. The smaller deviation from the optimal plan length for $h^{spectral}$ may be attributed to the increased amount of search performed by A*, as is evident by the large heuristic calls required for the more difficult problems. This suggests that $h^{spectral}$ may be providing underestimates in comparison to $h^{spatial}$. The coverage for both $h^{spatial}$ and $h^{spectral}$ started to decrease for problems with 10 blocks to around 8/10. The average coverage was 0.8 for h^{max}, 1 for h^{add}, 0.98 for LM-Cut, 0.97 for $h^{spatial}$, and 0.95 for $h^{spectral}$.

Interestingly, $h^{spatial}$ requires less heuristic calls on average than h^{add}, yet finds substantially cheaper plans. This suggests that STRIPS-HGNs are able to learn a form of generalisable knowledge which may be effectively applied across Blocksworld problems with a varying number of blocks.

Matching Blocksworld

Figure 5.14 shows our results for the Matching Blocksworld experiment described in Section 5.2.2. Notice that $h^{spatial}$ requires substantially less heuristic calls than h^{max} for all problems, and LM-cut for the more difficult problems. Although $h^{spatial}$ requires slightly more heuristic calls than h^{add}, the deviation from the optimal plan length for $h^{spatial}$ is much less than the deviations for h^{add}. This suggests that a STRIPS-HGN is able to learn some knowledge on how the polarities of a gripper interact with a block.

In terms of CPU time, $h^{spatial}$ is not very competitive with the other heuristics given the cost of message passing and our sub-optimal implementation. In our experiments, we observed that $h^{spatial}$ begins to struggle for problems where the goal is to stack 8 blocks into a single tower. For such problems, the coverage hovered around 4/10. This decreased performance may be attributed to the substantial time required to compute a single heuristic value (0.01 to 0.02 seconds), which means that A* will time out before a plan is found. The average coverage was 0.85 for h^{max}, 1 for h^{add}, 0.98 for LM-Cut, and 0.83 for $h^{spatial}$.

Gripper

The results of our Gripper experiments for both spectral and spatial-based HGNs are shown in Figure 5.15. We can see that $h^{spatial}$ is able to solve all the test problems with

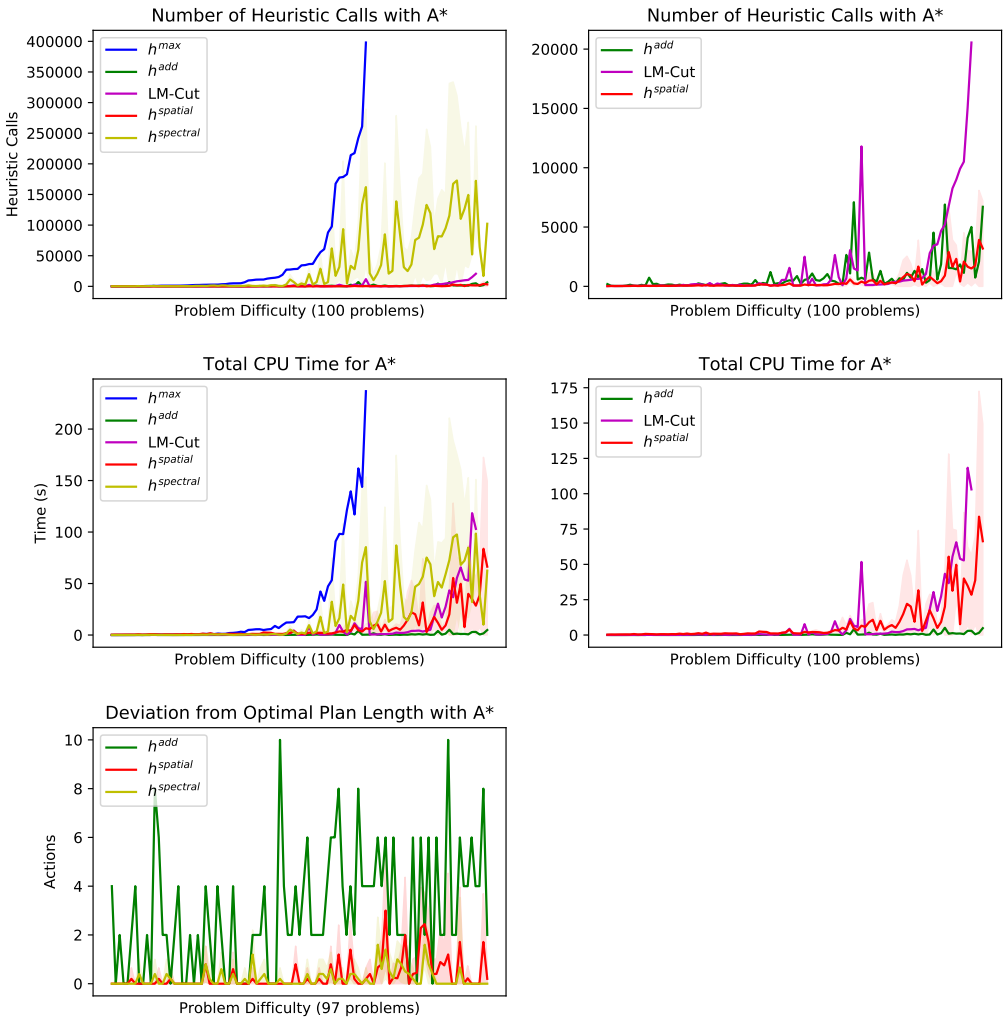


Figure 5.13: Results for Blocksworld. h^{max} and $h^{spectral}$ are removed from the plots in the second column.

4 to 20 balls. In comparison, none of the other heuristics are able to solve problems with more than 13 balls.

The number of heuristic calls required by $h^{spatial}$ does not exponentially increase as the problem size gets larger. In comparison, $h^{spectral}$ requires more heuristic calls on average than all the other heuristics we evaluated Gripper on. This experiment clearly exhibits the limited generalisation capability of spectral-based HGNs compared to that of spatial-based HGNs. However, for the problems $h^{spectral}$ was able to solve, its deviation from the optimal plan length was less than that for $h^{spatial}$. This suggests that the STRIPS-HGN for $h^{spatial}$ may be slightly overfitting to the training data with 1 to 3 balls, rather than learning generalisable knowledge for Gripper.

The average coverage was 0.59 for h^{max}, 0.59 for h^{add}, 0.41 for LM-Cut, 0.95 for $h^{spatial}$, and 0.48 for $h^{spectral}$. We observed that the coverage of $h^{spatial}$ drops to 16/20 for the most difficult problem with 20 balls.

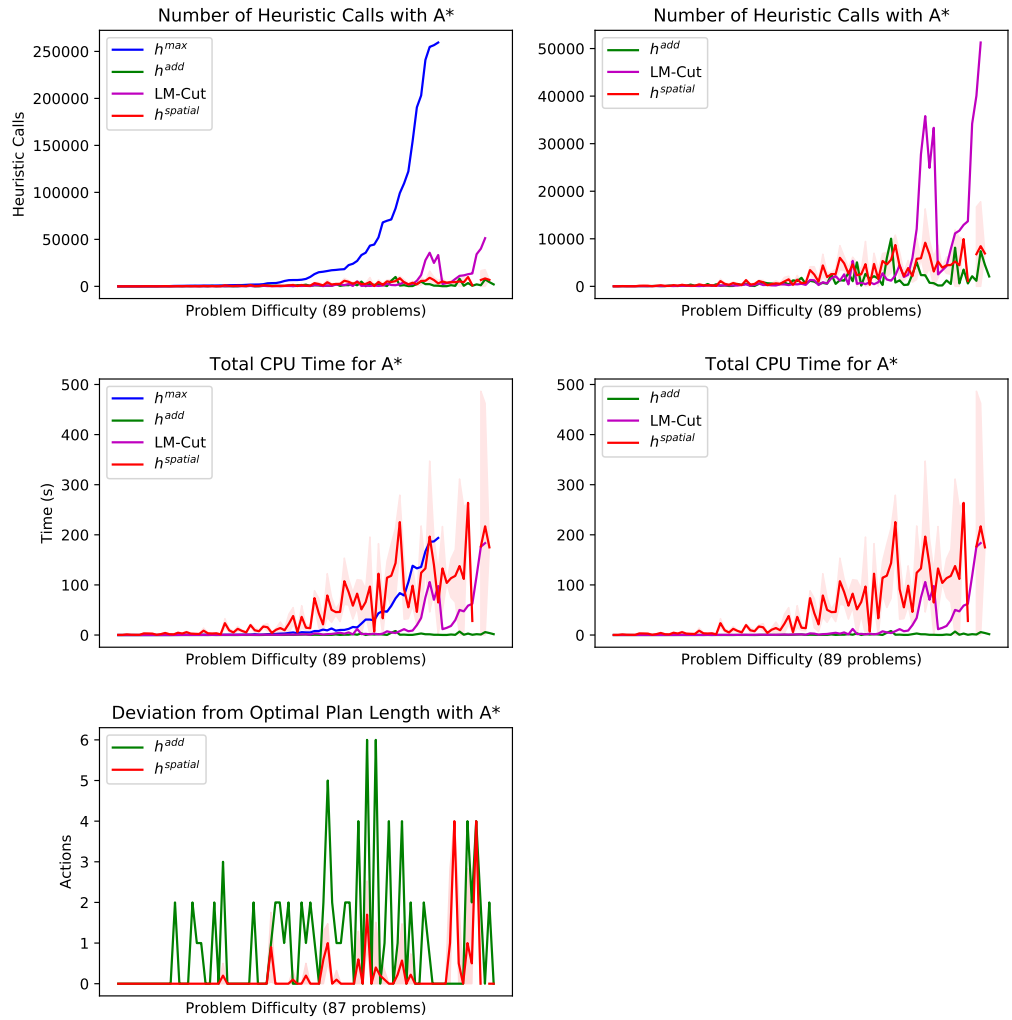


Figure 5.14: Results for Matching Blocksworld. The plots in the 2nd column remove h^{max}.

Moreover, it is interesting to note that across some experiments, the heuristic estimates computed by $h^{spatial}$ seem to be scaled by a large constant as the problem size increases. For example, in one experiment, $h^{spatial}$ gave a heuristic estimate of 510664 for the initial state in the problem with 20 balls. This strongly suggests that although $h^{spatial}$ may seem to be providing nonsensical values, it still provides an informative ranking of the states which helps A* find a plan in a relatively small number of heuristic calls. Additionally, it may be fruitful to evaluate our heuristics on greedy best-first search which considers the ranking of states given by our heuristic estimates.

Hanoi

Hanoi is a domain where a STRIPS-HGN is unable to learn anything substantial and $h^{spatial}$ subsequently provides uninformative heuristic estimates. This is depicted in

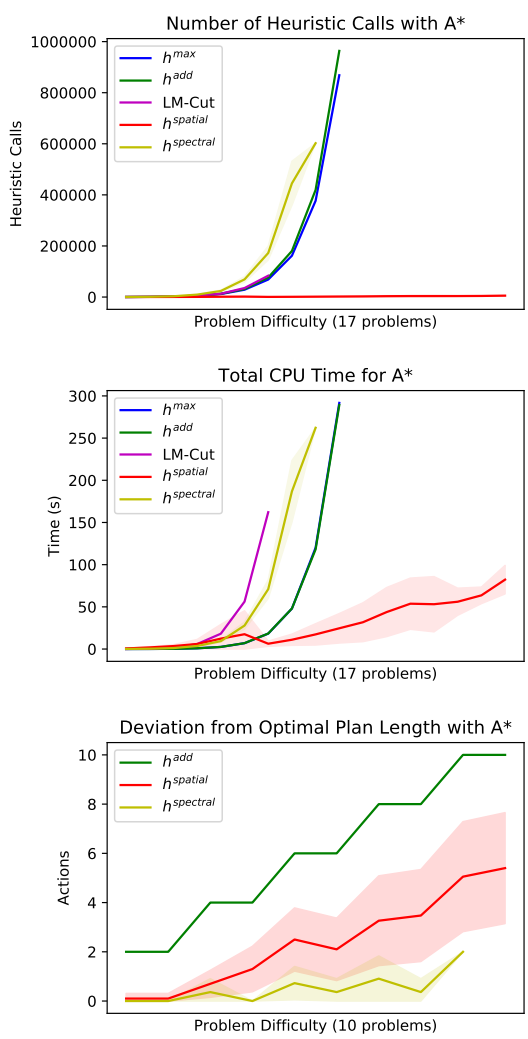


Figure 5.15: Results for Gripper.

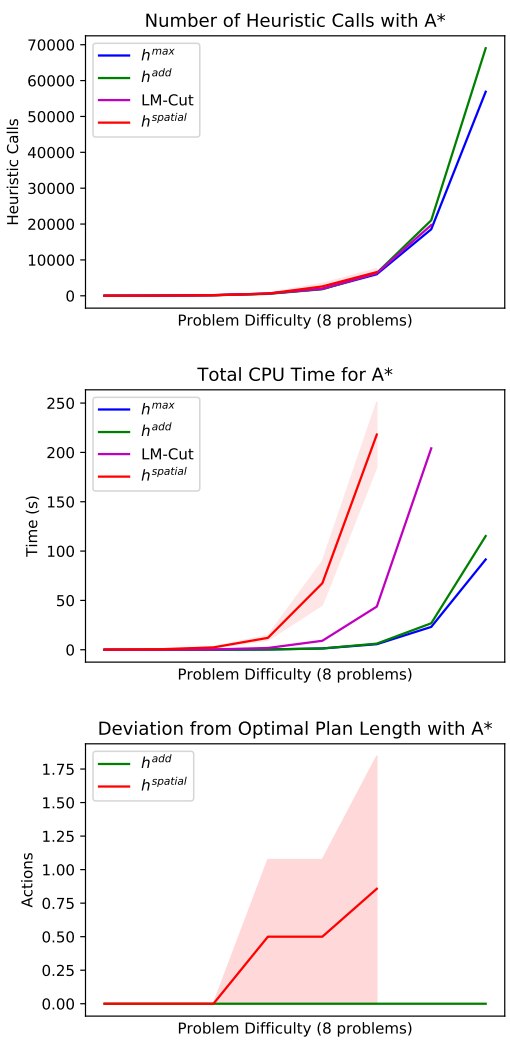


Figure 5.16: Results for Hanoi.

Figure 5.16, where the curve for $h^{spatial}$ stops very early in comparison to the other heuristics. The average coverage was 1 for h^{max} and h^{add}, 0.88 for LM-Cut, and 0.7 for $h^{spatial}$.

We can observe that that the deviation from the optimal plan length for $h^{spatial}$ is significantly more than the deviation of h^{add} which is 0. This could suggest that the network has overfitted to the problems in the training set. The more likely explanation is that STRIPS-HGNs are unable to learn the recursive relationship required to easily solve Hanoi problems.

Ferry

Figure 5.17 shows the results of the Ferry experiment we described in Section 5.2.5. Both h^{max} and LM-cut struggle significantly on the test problems, as indicated by the discontinuities of their respective curves.

On the other hand, $h^{spatial}$ is able to achieve coverage for all problems, and results in substantially less heuristic calls than h^{max} and LM-cut. We observed that the coverage of $h^{spatial}$ for the most difficult problem with 10 locations and 20 cars was 3/10. Additionally, the average coverage was 0.36 for h^{max}, 1 for h^{add}, 0.47 for LM-Cut, and 0.77 for $h^{spatial}$.

In line with the fact that the number of heuristic calls and CPU time for $h^{spatial}$ is more than that required by h^{add}, the deviations from the optimal plan length for $h^{spatial}$ is noticeably less than the deviations for h^{add}. To summarise, $h^{spatial}$ is able to generalise to much larger problems than the STRIPS-HGN was trained on. However, the performance of these trained networks may vary significantly as shown by the confidence bounds, suggesting a more suitable training procedure should be adopted.

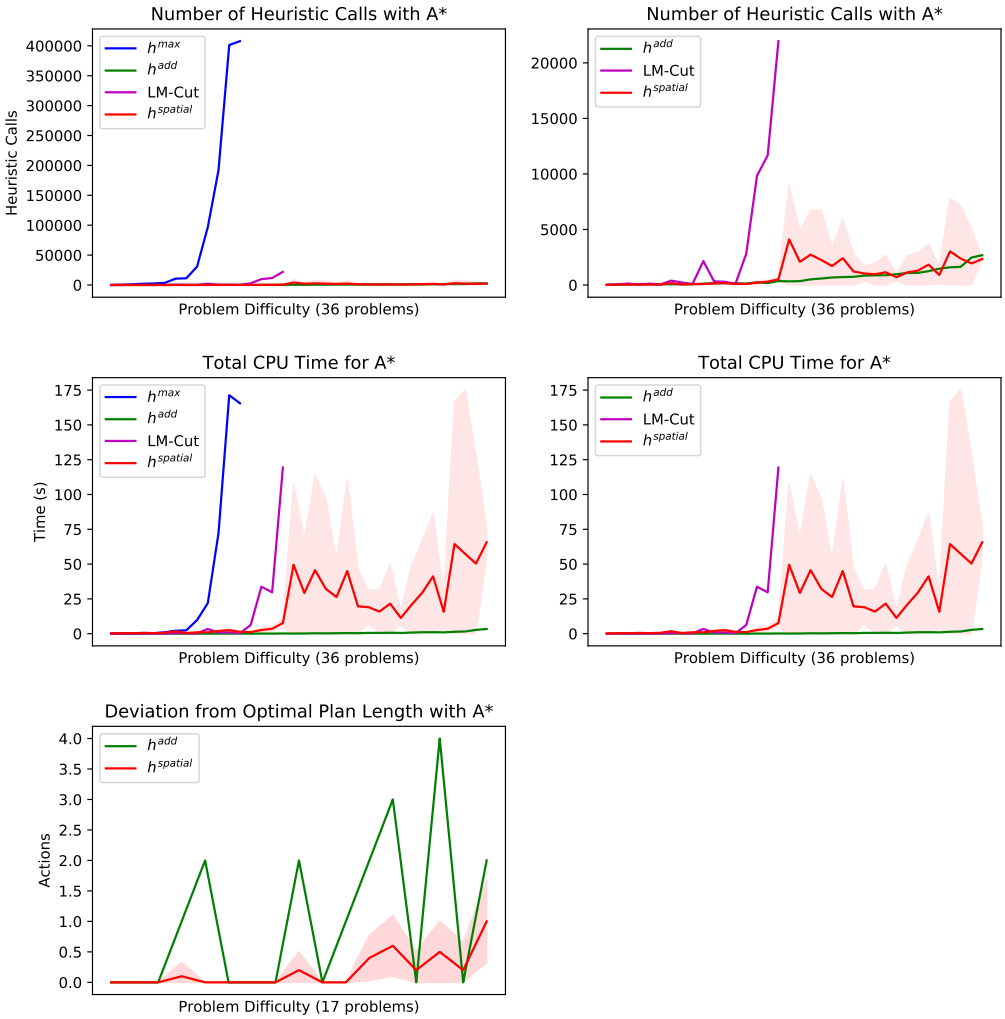


Figure 5.17: Results for Ferry. h^{max} is removed from the plots in the second column.

Zenotravel

The results for our Zenotravel experiments, which were described in Section 5.2.6, are shown in Figure 5.18. Firstly, we may observe that $h^{spatial}$ significantly outperforms h^{max} in terms of number of heuristic calls. As the problems become more difficult, both $h^{spatial}$ and LM-cut are unable to find a plan in the limited search time. In comparison h^{add} is able to find a near-optimal plan extremely quickly.

Despite this, the results of this experiment show that STRIPS-HGNs are able to learn some form of knowledge which helps it generalise to problems of larger size, and achieve a smaller deviation from the optimal plan length in comparison to h^{add}. Moreover, the number of heuristic calls required for $h^{spatial}$ is comparable to the heuristic calls required for LM-cut. The large spike in the confidence interval in the plots for $h^{spatial}$ arise from the problems where $h^{spatial}$ achieved low coverage (e.g. 2/10 coverage). The average coverage was 0.55 for h^{max}, 1 for h^{add}, 0.82 for LM-Cut, and 0.71 for $h^{spatial}$.

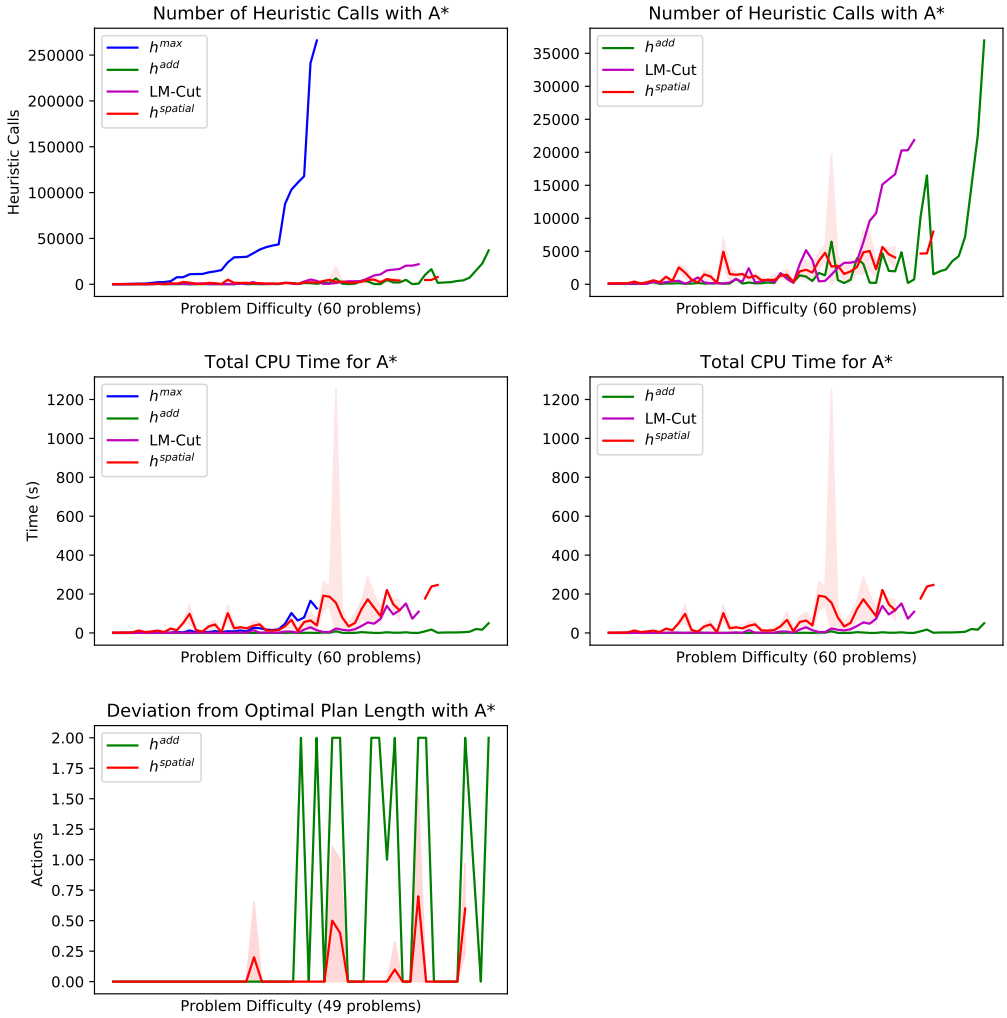


Figure 5.18: Results for Zenotravel. The discontinuities in a curve indicate problems which were unsolvable. h^{max} is removed from the plots in the second column.

5.3.3 Learning Domain-Independent Heuristics

Training and Evaluating on Blocksworld, Zenotravel and Gripper

The results of our multi-domain experiments are shown in Figure 5.19 for Blocksworld, Figure 5.20 for Zenotravel, and Figure 5.21 for Gripper. We may observe that $h^{spatial}$ is able to generalise to larger problems across all three domains. The results we achieved for our multi-domain experiments are, in general, similar to the results we achieved in the domain-dependent experiments presented in Section 5.3.2.

For Blocksworld, the domain-independent heuristic we learn performs marginally worse in terms of number of heuristic calls and CPU time than the domain-dependent heuristic we learn (compare Figures 5.13 and 5.19). However, the average coverage for both approaches is extremely similar (0.95 for domain-dependent, 0.97 for domain-independent). Additionally, the domain-independent heuristic achieves a lower deviation from the optimal plan length. This may be attributed to the increased number of nodes explored by A* with the domain-independent heuristic which is not as 'informative' as the domain-dependent heuristic in terms of actual heuristic values.

For Zenotravel, the domain-independent heuristic requires more heuristic calls and CPU time than its equivalent domain-dependent heuristic (compare Figures 5.13 and 5.20). The domain-dependent heuristic achieves a slightly better average coverage of 0.71 compared to 0.6 for the domain-independent heuristic. Both heuristics achieve similar deviations from the optimal plan lengths.

For Gripper, the deviation from the optimal plan length for the domain-independent heuristic we learned is surprisingly less than the deviation for the domain-dependent heuristic we learned (compare Figures 5.15 and 5.21). This suggests that the domain-dependent heuristic may have been overtrained, as we have shown it is possible to learn a domain-independent heuristic that results in plans of lower cost. The average coverage for the domain-dependent heuristic was 0.95, and 0.69 for the domain-independent heuristic. This suggests that the multi-domain STRIPS-HGN has learned some knowledge which, although leads to shorter plans, is not sufficient to generalise reliably across to larger problems.

Training on Zenotravel and Gripper, and evaluating on Blocksworld

Figure 5.22 shows the results of our multi-domain experiment where we trained STRIPS-HGNs on Zenotravel and Gripper, and evaluated the networks on Blocksworld problems with 4-8 blocks. Evidently, $h^{spatial}$ is able to generalise to all the test problems and requires fewer heuristic calls than h^{max}. The average coverage was 1 for h^{max}, h^{add} and LM-Cut, and 0.87 for $h^{spatial}$. The reduced average coverage for $h^{spatial}$ arises from the fact that it begins to struggle with problems with 8 blocks where it achieves an average coverage of 0.5.

The network seems to have learned a general procedure for aggregating vertex (proposition) and hyperedge (action) features despite the fact that the latent representations of these features are likely not to be informative to Blocksworld.

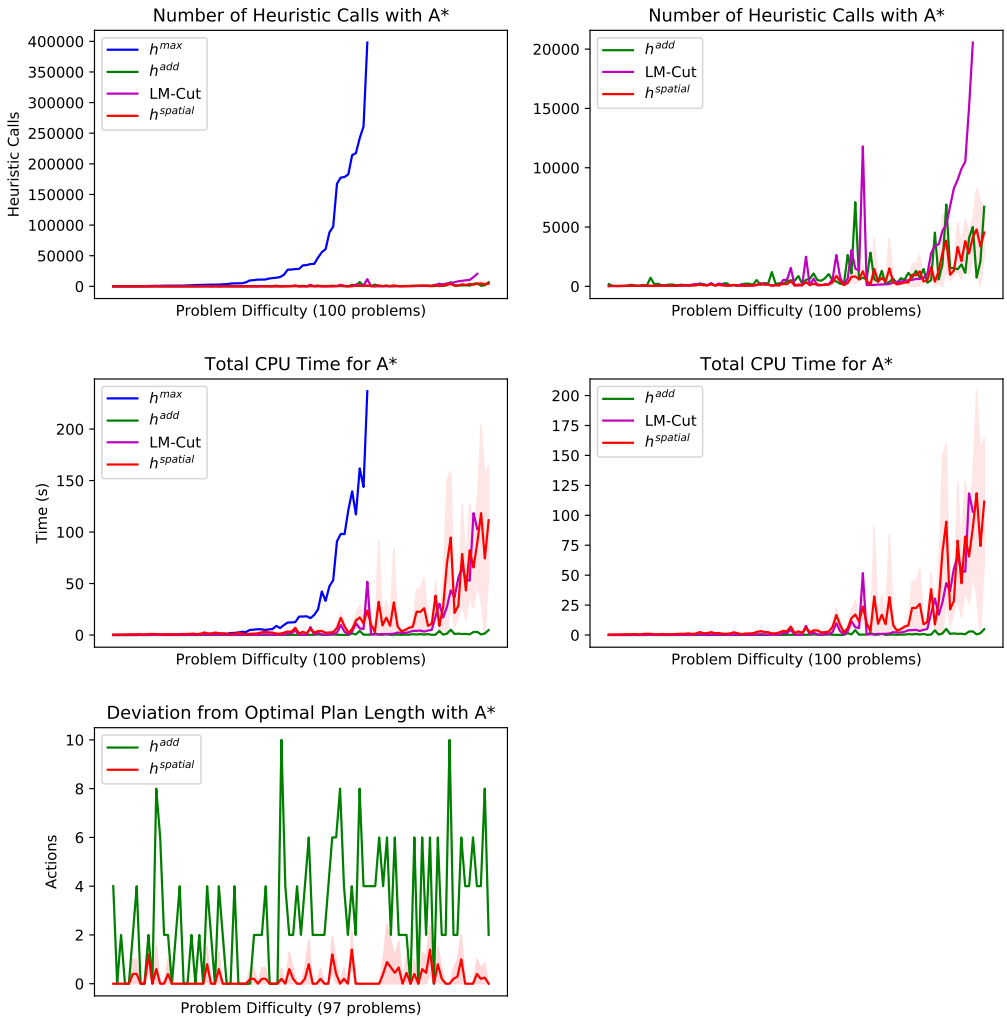


Figure 5.19: Results for Multi-Domain Blocksworld. h^{max} is removed from the plots in the second column.

We observed that the heuristic values provided by $h^{spatial}$ for the Blocksworld problems very occasionally hovered between -1 and 1. Despite this, the performance of A* with $h^{spatial}$ suggests that the network is able to provide an informative ranking for each state which helps it find plans with similar or lower costs than h^{add}.

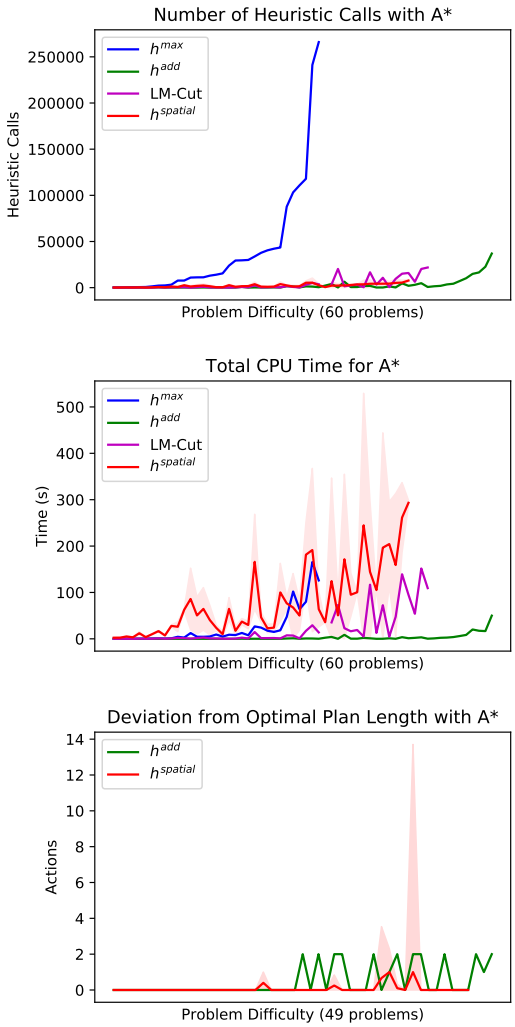


Figure 5.20: Results for Multi-Domain Zenotravel.

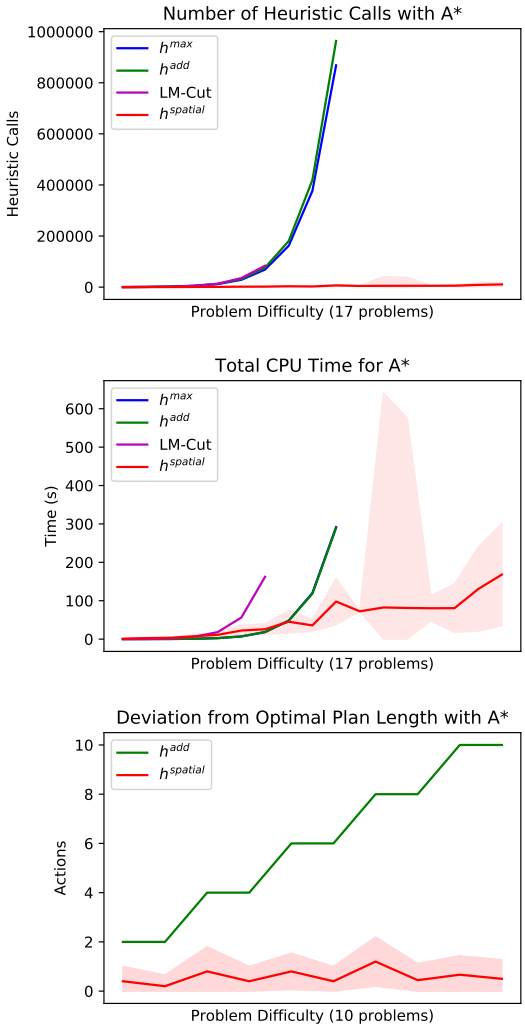


Figure 5.21: Results for Multi-Domain Gripper.

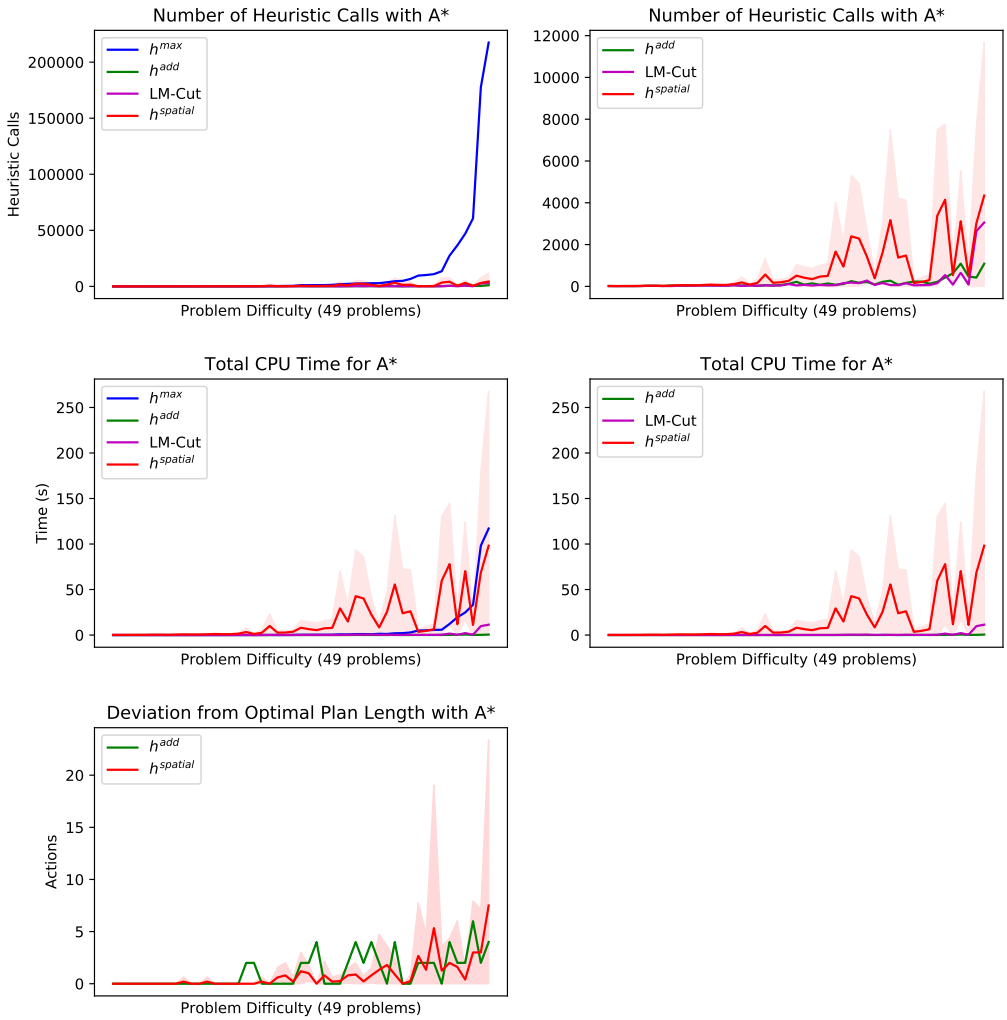


Figure 5.22: Results for Blocksworld where our STRIPS-HGNs were trained on Zenotravel and Gripper. Problem `bw-8-3326-7` is not shown due to the large confidence interval given by the low coverage for $h^{spatial}$, which skews the plots and makes it difficult to read the y-axes. h^{max} is not shown in the plots for the 2nd column.

5.3.4 Discussion

Our experimental results show that we are able to train STRIPS-HGNs to learn three classes of heuristics: problem-size dependent, domain-dependent, and domain-independent heuristics.

Through our experiments for Sokoban, Blocksworld and Gripper, we have shown that the generalisation capability of spectral-based HGNNs is limited. We would expect this limitation to be extended to other planning domains, as HGNNs only support undirected hypergraphs and perform convolutions using the normalised incidence matrix which is an approximation of the hypergraph Laplacian.

In contrast, it is possible to train STRIPS-HGNs to learn domain-dependent and domain-independent heuristics in a variety of domains. We showed that a network trained on a set of small-sized problems is able to generalise to problems of much larger sizes. Moreover, we showed it is possible for a STRIPS-HGN to provide informative heuristic estimates for states in a domain it was not trained on. We believe the promising performance of STRIPS-HGNs may be traced back to our recurrent encoding-processing-decoding architecture, which allows messages to be passed between the vertices and hyperedges in the hypergraph in a rich latent feature space.

Unlike existing approaches in deep learning for planning which learn which actions to apply in a given state [Toyer et al., 2019; Issakkimuthu et al., 2018; Garg et al., 2019], the heuristics we learn can be easily combined with any search algorithm to provide formal guarantees. For example, using A* with $h^{spatial}$ guarantees that a plan will be found if one exists. Although it is possible to combine an Action Schema Network with Monte-Carlo Tree Search [Shen et al., 2019], learning heuristics allow us to reason at a much lighter level.

Moreover, we have demonstrated that STRIPS-HGNs are able to learn informative heuristics from simple features derived from a STRIPS problem. This is in contrast to [Yoon et al., 2008] who learn heuristic functions based on the features extracted from the relaxed plan of a problem, and [Gomoluch et al., 2017] who compute improvements on h^{FF} using MLPs (i.e., h^{FF} is fed as input to the network).

Our work takes an important first step in investigating whether STRIPS-HGNs are feasible for learning heuristics. Nevertheless, there are many avenues for future work such as speeding up the time required to compute a heuristic estimate, and extending our STRIPS-HGNs to probabilistic planning. We provide a detailed discussion of potential future work in Section 6.2.

Chapter 6

Conclusion

In this chapter, we summarise the main contributions we have made in this thesis, and discuss promising directions for future research which aim to improve the performance and generalisation power of STRIPS-HGNs.

6.1 Contributions

The main objective of this thesis has been to investigate the feasibility of applying deep learning for learning heuristics in planning. We have shown that STRIPS-HGNs are able to learn problem-size dependent, domain-dependent and domain-independent heuristics by exploiting the hypergraph structure of a delete relaxed STRIPS problem. By training a STRIPS-HGN on a small set of problems, the network is able to generalise and compute informative heuristic estimates for larger problems it was not trained on. Our work has made three key contributions:

1. **Hypergraph Networks**
   Chapter 3 introduced Hypergraph Networks (HGNs), our novel framework which generalises Graph Networks [Battaglia et al., 2018] to hypergraphs. A Hypergraph Network is composed of HGN blocks, which are hypergraph-to-hypergraph functions. A HGN block is designed to be extremely flexible, as we may implement each update function and aggregation function in any manner as long as they satisfy the input and output requirements of the block, and the restrictions imposed by the HGN framework (i.e., aggregation functions must be permutation invariant to the inputs). Moreover, we discussed how existing state-of-the-art deep learning models for hypergraphs may be represented using HGNs. Hypergraph Networks provides an important framework for comparing and understanding the differences between existing hypergraph learning models, as well as providing an architecture for future research into designing more powerful models which fully exploit the features and structure of a hypergraph.

2. **STRIPS-HGNs: a Hypergraph Network for Learning Heuristics**
   STRIPS-HGNs is an instance of a Hypergraph Network which is designed to learn heuristics by approximating shortest paths over the hypergraph induced by the delete relaxation of a STRIPS problem. Each vertex in the hy-

pergraph represents a proposition, while each hyperedge represents an action. A STRIPS-HGN uses a recurrent encode-process-decode architecture to incrementally propagate the latent vertex and hyperedge features by using message passing.

STRIPS-HGNs are very flexible as they do not impose any rules on the input features for each vertex and hyperedge, and are agnostic to the implementation of each update and aggregation function. In fact, we showed how STRIPS-HGNs may be modified to learn which action to apply in a state rather than compute a heuristic estimate.

The inherent design of a STRIPS-HGN supports combinatorial generalisation, as the identical core HGN block is repeatedly applied to the latent hypergraph – this is analogous to message passing. Hence, the receptive field of a STRIPS-HGN may be increased or decreased by scaling the number of message passing steps. This is a significant advantage over existing deep learning models for planning, such as Action Schema Networks [Toyer et al., 2019], which have a fixed receptive field that is determined by the number of hidden layers.

The major limitation of STRIPS-HGNs is that we cannot provide any guarantees that the learned heuristic is admissible, as it is unfeasible to understand what a network is exactly computing. Nevertheless, our results show that the heuristic $h^{spatial}$ learned by a STRIPS-HGN is able to outperform h^{add}, h^{max} and LM-cut for certain domains.

3. **Extensive Empirical Evaluation**
We firstly provided an extremely detailed description of our experimental setup, which included our training procedure which performs binning and stratified k-Fold to reduce any potential noise and demonstrate robustness over the training set used. Next, we trained and evaluated both spectral-based Hypergraph Neural Networks and spatial-based STRIPS-HGNs on a variety of domains. Our results demonstrated the weak generalisation capability of a spectral-based HGNN to problems larger than the problems the network was trained on. We observed that the only advantage of spectral-based HGNNs over STRIPS-HGNs is that it is computationally cheaper to train and evaluate.

   Our experiments show that STRIPS-HGNs are able to learn problem-size dependent, domain-dependent, and domain-independent heuristics. We showed that the heuristic $h^{spatial}$ learned by a STRIPS-HGN generalises to problems of much larger size than the problems the network was trained on. For all domains except for Hanoi, the heuristic $h^{spatial}$ learned by a STRIPS-HGN requires substantially less heuristic calls than h^{max} and yet provides near-optimal plans. In general, $h^{spatial}$ required a similar number of heuristic calls to LM-cut, but in certain experiments including Blocksworld and Gripper, $h^{spatial}$ required less heuristic calls than all the baselines h^{max}, h^{add} and LM-cut.

   Unfortunately, $h^{spatial}$ is not very competitive with the baseline heuristics in terms of CPU time due to our sub-optimal implementation and the unavoidable

cost of message passing in a large hypergraph. Despite this, our results show that it is feasible for a STRIPS-HGN to learn knowledge which helps it compute informative heuristic estimates, and hence opens up a wealth of potential directions for future research.

6.2 Future Work

Now, we discuss several promising directions for future research which range from improving the planning performance of STRIPS-HGNs, to extending STRIPS-HGNs to probabilistic planning.

6.2.1 Speeding up a STRIPS-HGN

Hypergraph Pruning

One limitation of STRIPS-HGNs is that the time required to compute a heuristic increases as the number of vertices and hyperedges in a hypergraph increase. This may be attributed to our message passing scheme, which ensures that messages are passed between all vertices and hyperedges in the hypergraph.

Furthermore, STRIPS-HGNs require the entire hypergraph structure of a delete relaxed STRIPS problem to be passed as input to a network. This is in contrast to h^{max} and h^{add}, whose implementations implicitly generate intermediate subsets of the hypergraph as they incrementally compute a heuristic value. This suggests that the entire hypergraph structure may not be required as input to STRIPS-HGNs, especially since it is only feasible to perform a limited number of message passing steps.

Future work could investigate whether it is possible to prune extraneous propositions and actions from a problem to reduce the size of its hypergraph – this may substantially decrease the time required by a STRIPS-HGN to compute a heuristic estimate. Moreover, it may be possible to restrict message passing to a subset of vertices and hyperedges in the hypergraph. We could determine this subset by analysing which vertices and hyperedges which have received a 'signal' from the vertices in the initial or goal state. Such an approach would ideally be able to adaptively scale to the size of the hypergraph based on the number of message passing steps we use.

Implementation-based Optimisations

As we mentioned in Section 5.1, our current implementation of Hypergraph Networks is not optimal, and hence may be sped up. A more optimised implementation would result in a reduction in the CPU time required to compute a single heuristic estimate, which currently hovers around 0.01 to 0.02 seconds for a hypergraph with several hundred vertices and hyperedges.

The evaluation of our learned heuristics and baseline heuristics should also be performed in Fast Downward [Helmert, 2006] rather than Pyperplan [Alkhazraji et al., 2011]. Fast Downward is a state-of-the-art classical planner, while Pyperplan

is designed to be more of a teaching tool rather than the best planner. Evidently, we found that the implementations of h^{add}, h^{max} and LM-cut in Pyperplan are several times slower than their counterparts in Fast Downward.

Additionally, we could investigate how we may exploit multiple CPU cores or even a GPU to parallelise the tensor operations required in the HGN blocks within a STRIPS-HGN. Doing so would require batching together operations to ensure the overheads of multiprocessing do not outweigh the benefits of parallel computations. Ideally, we would observe a speed-up in both training and evaluation time as a result of using more powerful hardware.

6.2.2 Improving the performance of STRIPS-HGNs

Similar to all deep learning models, there are countless hyperparameters in a STRIPS-HGN which may be tuned including the number of message passing steps, the implementation of each update and aggregation function, the hidden dimensionalities of vertices and hyperedges, etc. We outline potential future work which we believe may vastly improve the performance of STRIPS-HGNs below.

Master Vertex

The receptive field of a STRIPS-HGN is effectively limited by the number of message passing steps we perform. This means that if the number of message passing steps is less than the optimal heuristic value for a given state, then signals from the initial propositions are unable to reach all the goal propositions.

Gilmer et al. [2017] propose a potential solution to this problem by creating a new “master" vertex which is connected to every other vertex in the hypergraph with a special hyperedge type. This master vertex, which would have a very high feature dimensionality, would act as a global scratch space for vertices to read and write to in each step of message passing. This would in theory, allow signals to “travel long distances" even for a small number of message passing steps [Gilmer et al., 2017].

We expect that a STRIPS-HGN with a master vertex will be able to provide more accurate heuristic estimates than a vanilla STRIPS-HGN. Although the master vertex approach would be more computationally expensive to train and evaluate, this increased time may pay off if the heuristic estimates are vastly better than the estimates of a vanilla STRIPS-HGN. Subsequently, this would allow a search algorithm find a solution in less CPU time and heuristic calls.

Proposition and Action Features

The networks in our experiments use very simple features derived from a STRIPS problem. It may be possible to improve the generalisation capability and planning performance of a STRIPS-HGN by using a set of richer input features.

We could include fact landmarks as features for propositions and action landmarks as features for actions in a hypergraph – these would be computed using LM-cut [Helmert and Domshlak, 2009]. This approach would be similar to Action

Schema Networks, which use action landmarks to increase a network's theoretical receptive field and improve its generalisation ability. Alternatively, another option could be to supplement a STRIPS-HGN with a heuristic value calculated by another heuristic such as LM-Cut or h^{FF} [Hoffmann, 2001]. The resulting problem would be equivalent to learning an improvement over these heuristics, rather than learning a heuristic from scratch.

Although using features computed by a different planning heuristic arguably defeats the purpose of learning a heuristic based solely from the structure of the hypergraph, the heuristics learned from a richer set of input features may lead to improved planning performance and generalisation across problem size and domains.

Improved Training Procedure

Firstly, our current training procedure is not very efficient as we not only train a separate network for each fold given by stratified k-Fold, but also only select one of these networks for evaluation. This means that precious training time is discarded in an effort to improve model stability. Moreover, we observed that the performance of a STRIPS-HGN may vary significantly across multiple runs of the same experiment. This behaviour could be attributed to a poor choice of hyperparameters, such as a learning rate that is too high or a batch size that is too small.

We believe there is definitely room for improvement in our training procedure. These improvements could involve defining a more sophisticated algorithm which splits the training data in a consistent manner that guarantees each partition is an accurate representation of the dataset. A more reasoned study should also be carried out to determine how the choice of hyperparameters for a STRIPS-HGN and the training procedure may affect the performance of the network.

Guaranteeing the Invariance of STRIPS-HGNs

In Section 5.1, we argued that STRIPS-HGNs with Multilayer perceptrons (MLP) as the update functions are invariant to the renaming of objects and actions, but are **not invariant** to the renaming of the predicates (propositions) in the domain definition. This is because MLPs are not permutation invariant to the ordering of the inputs – that is, different orderings of the same input give different outputs. The alphabetical sorting procedure we imposed on the proposition names of the vertices may lead to an altered ordering if we rename the predicates in the domain definition.

Invariance is an important property of any heuristic, as it guarantees that identical heuristic values are returned for isomorphic states in a set of isomorphic problems. Future work should investigate a suitable implementation for each update function which guarantees permutation invariance to the ordering of the inputs. This could be achieved through DeepSets [Zaheer et al., 2017], or a fixed transform matrix which outputs the same weights for vertex features regardless of their ordering [Jiang et al., 2019].

6.2.3 Extending STRIPS-HGNs beyond STRIPS problems

Finite Domain Representation (FDR) Planning

Steinmetz and Torralba [2019] showed that it is possible to generalise abstraction and critical-path heuristics to hypergraphs for FDR planning tasks. Their work introduced *hyperabstractions*, heuristics which approximate goal distances based on the fundamental ideas behind abstraction and critical-path heuristics.

It may be possible to learn better approximations of the shortest path over the *B-hypergraphs* and *F-hypergraphs* they define by using Hypergraph Networks. This would require extending STRIPS-HGNs to support multi-valued propositions for each vertex.

Probabilistic Planning

Probabilistic planning is an extension of classical planning where actions have stochastic outcomes. It is possible to apply delete-relaxation heuristics defined for classical planning to probablistic planning through determinisation. Determinisation relaxes a probabilistic problem into a deterministic one by either choosing a single outcome for each probabilistic action (single-outcome determinisation), or decomposing each outcome of a probabilistic action into a separate deterministic action (all-outcomes determinisation) [Keller and Eyerich, 2011]. Evidently, heuristics which rely on determinisation ignore the true probabilities of outcomes and unintended side-effects of the probabilistic actions when computing a heuristic value. Instead, the determinisation of the problem allows the heuristic to choose the most convenient path. Current state-of-the-art heuristics in probabilistic planning which **do not** perform determinisation, such as h^{pom} and h^{roc} [Trevizan et al., 2017], are difficult to construct and can be expensive to compute.

It is possible to model computing a heuristic estimate in probabilistic planning as approximating the shortest path over the hypergraph induced by a Stochastic Shortest Path problem (SSP) [Bertsekas and Tsitsiklis, 1991]. We may represent the hypergraph induced by a SSP as the collection of hyperedges defined by the scheme depicted in Figure 6.1 – vertices in the hypergraph now represent propositions or actions, while hyperedges represent an aggregation of preconditions or individual stochastic outcomes.

We may redesign STRIPS-HGNs to support this hypergraph structure, and train a network to learn an aggregation of propositions conditioned on the probabilities of each action. This would hopefully provide more accurate heuristic estimates than determinisation-based heuristics.

6.3 Final Remarks

We have introduced Hypergraph Networks, a framework for designing deep learning models which operate over hypergraphs. Using HGNs, we have successfully shown that it is feasible to learn heuristics for planning with STRIPS-HGNs. Our work

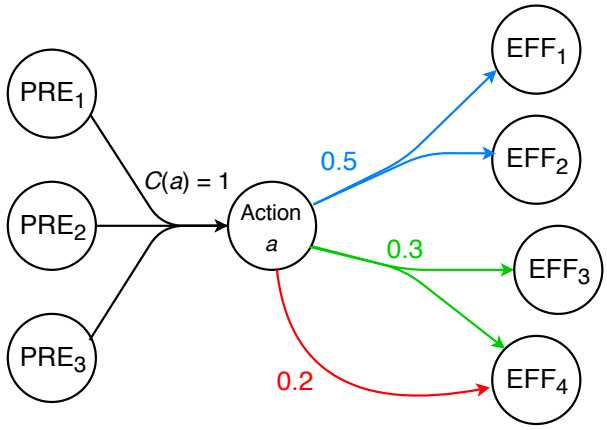


Figure 6.1: Example of a potential hyperedge representation for a probabilistic action a with 3 preconditions, 3 stochastic outcomes, and a cost of $C(a) = 1$ in a SSP. Note that the action is now an explicit vertex which 'receives' preconditions, and 'sends' stochastic outcomes.

represents an important first step in learning heuristics which are able generalise across problems of different sizes and across different domains.

We have discussed multiple potential directions for future work which aim to additionally improve the planning performance of STRIPS-HGNs and extend them beyond classical planning to probabilistic planning. We hope that this thesis further bridges the gap between planning and machine learning.

List of Figures

List of Tables

Bibliography

Alkhazraji, Y.; Frorath, M.; Grützner, M.; Liebetraut, T.; Ortlieb, M.; Seipp, J.; Springenberg, T.; Stahl, P.; and Wülfing, J., 2011. Pyperplan. https://bitbucket.org/malte/pyperplan. (cited on pages 54 and 89)

Arfaee, S. J.; Zilles, S.; and Holte, R. C., 2010. Bootstrap learning of heuristic functions. In *Third Annual Symposium on Combinatorial Search*. (cited on page 16)

Bai, S.; Zhang, F.; and Torr, P. H., 2019. Hypergraph convolution and hypergraph attention. *arXiv preprint arXiv:1901.08150*, (2019). (cited on page 22)

Bajpai, A. N.; Garg, S.; and Mausam, 2018. Transfer of deep reactive policies for mdp planning. In *Advances in Neural Information Processing Systems*, 10965–10975. (cited on page 16)

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*, (2018). (cited on pages vii, 3, 4, 12, 13, 19, 28, 29, 30, 31, 32, 33, 37, 44, 47, 50, 87, 95, 96, and 99)

Bertsekas, D. P. and Tsitsiklis, J. N., 1991. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16, 3 (1991), 580–595. (cited on page 92)

Boddy, M.; Gohde, J.; Haigh, T.; and Harp, S., 2005. Course of action generation for cyber security using classical planning. In *Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'05 (Monterey, California, USA, 2005), 12–21. AAAI Press. http://dl.acm.org/citation.cfm?id=3037062.3037065. (cited on page 1)

Bonet, B. and Geffner, H., 2001. Planning as heuristic search. *Artificial Intelligence*, 129, 1-2 (2001), 5–33. (cited on pages 7 and 40)

Boser, B. E.; Guyon, I. M.; and Vapnik, V. N., 1992. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, 144–152. ACM. (cited on page 17)

Bresina, J. L.; Jónsson, A. K.; Morris, P. H.; and Rajan, K., 2005. Activity planning for the mars exploration rovers. In *International Conference on International Conference on Automated Planning and Scheduling*. (cited on page 1)

Bruna, J.; Zaremba, W.; Szlam, A.; and LeCun, Y., 2013. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, (2013). (cited on page 13)

Bylander, T., 1994. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69, 1-2 (1994), 165–204. (cited on page 6)

Chan, T.-H. H.; Louis, A.; Tang, Z. G.; and Zhang, C., 2018. Spectral properties of hypergraph laplacian and approximation algorithms. *J. ACM*, 65, 3 (Mar. 2018), 15:1–15:48. doi:10.1145/3178123. http://doi.acm.org/10.1145/3178123. (cited on page 23)

Culberson C., J., 1997. Sokoban is pspace-complete. Technical report. (cited on page 69)

Defferrard, M.; Bresson, X.; and Vandergheynst, P., 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, 3844–3852. (cited on page 13)

Duvenaud, D. K.; Maclaurin, D.; Iparraguirre, J.; Bombarell, R.; Hirzel, T.; Aspuru-Guzik, A.; and Adams, R. P., 2015. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, 2224–2232. (cited on page 14)

Feng, Y.; You, H.; Zhang, Z.; Ji, R.; and Gao, Y., 2019. Hypergraph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 3558–3565. (cited on pages 19, 21, 22, 23, 59, and 95)

Fern, A.; Khardon, R.; and Tadepalli, P., 2011. The first learning track of the international planning competition. *Machine Learning*, 84, 1-2 (2011), 81–107. (cited on pages 64, 65, 68, and 69)

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 (1971), 189–208. (cited on pages 1 and 6)

Francès, G.; Corrêa, A. B.; Geissmann, C.; and Pommerening, F., 2019. Generalized potential heuristics for classical planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 5554–5561. (cited on page 9)

Gallo, G.; Longo, G.; Pallottino, S.; and Nguyen, S., 1993. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42, 2 (1993), 177 – 201. doi:https://doi.org/10.1016/0166-218X(93)90045-P. http://www.sciencedirect.com/science/article/pii/0166218X9390045P. (cited on pages 19, 21, and 95)

Garg, S.; Bajpai, A.; and Mausam, 2019. Size independent neural transfer for RDDL planning. In *Proceedings of the Twenty-Ninth International Conference on Automated*

Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019., 631–636. (cited on pages 16 and 86)

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T., 2016. Learning to rank for synthesizing planning heuristics. *arXiv preprint arXiv:1608.01302*, (2016). (cited on pages 17 and 48)

Geffner, H. and Bonet, B., 2013. A concise introduction to models and methods for automated planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8, 1 (2013), 1–141. (cited on pages 1, 5, 6, 7, and 9)

Geissmann, C., 2015. *Learning Heuristic Functions in Classical Planning*. Master's thesis, University of Basel. (cited on pages 16 and 17)

Ghallab, M.; Howe, A.; Knoblock, C.; Mcdermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D., 1998. PDDL—The Planning Domain Definition Language. Technical report. (cited on page 7)

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E., 2017. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 1263–1272. JMLR. org. (cited on pages 13, 14, 22, 28, 33, 34, 44, 45, and 90)

Gomoluch, P.; Alrajeh, D.; Russo, A.; and Bucchiarone, A., 2017. Towards learning domain-independent planning heuristics. *arXiv preprint arXiv:1707.06895*, (2017). (cited on pages 17, 42, and 86)

Goodfellow, I.; Bengio, Y.; and Courville, A., 2016. *Deep learning*. (cited on page 11)

Groshev, E.; Tamar, A.; Goldstein, M.; Srivastava, S.; and Abbeel, P., 2018. Learning generalized reactive policies using deep neural networks. In *2018 AAAI Spring Symposium Series*. (cited on pages 2, 15, and 16)

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*, (2018). (cited on page 43)

Hart, P. E.; Nilsson, N. J.; and Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4, 2 (July 1968), 100–107. doi:10.1109/TSSC.1968.300136. (cited on pages 7 and 8)

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; Koenig, S.; et al., 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. (cited on page 9)

Haslum, P. and Geffner, H., 2000. Admissible heuristics for optimal planning. In *Proceedings of the 5th International Conference of AI Planning Systems*. (cited on pages 8 and 41)

He, K.; Zhang, X.; Ren, S.; and Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, 1026–1034. IEEE Computer Society, Washington, DC, USA. doi:10.1109/ICCV.2015.123. http://dx.doi.org/10.1109/ICCV.2015.123. (cited on page 56)

Helmert, M., 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26 (2006), 191–246. (cited on pages 54 and 89)

Helmert, M., 2008. *Understanding planning tasks: domain complexity and heuristic decomposition*, vol. 4929. Springer. (cited on page 67)

Helmert, M. and Domshlak, C., 2009. Landmarks, critical paths and abstractions: what's the difference anyway? In *Nineteenth International Conference on Automated Planning and Scheduling*. (cited on pages 8, 9, and 90)

Hochreiter, S. and Schmidhuber, J., 1997. Long short-term memory. *Neural computation*, 9, 8 (1997), 1735–1780. (cited on page 2)

Hoffmann, J., 2001. Ff: The fast-forward planning system. *AI magazine*, 22, 3 (2001), 57–57. (cited on pages 17 and 91)

Iba, G. A., 1989. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3, 4 (1989), 285–317. (cited on page 68)

Issakkimuthu, M.; Fern, A.; and Tadepalli, P., 2018. Training deep reactive policies for probabilistic planning problems. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*. (cited on pages 16 and 86)

Jiang, J.; Wei, Y.; Feng, Y.; Cao, J.; and Gao, Y., 2019. Dynamic hypergraph neural networks. In *Proceedings of International Joint Conferences on Artificial Intelligence*. (cited on pages 25, 26, 27, 91, and 95)

Joulin, A.; Grave, E.; Bojanowski, P.; and Mikolov, T., 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, (2016). (cited on page 2)

Keller, T. and Eyerich, P., 2011. A polynomial all outcome determinization for probabilistic planning. In *Twenty-First International Conference on Automated Planning and Scheduling*. (cited on page 92)

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L., 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, 6348–6358. (cited on page 16)

Kingma, D. P. and Ba, J., 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, (2014). (cited on pages 50 and 61)

Kipf, T. N. and Welling, M., 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*. (cited on pages 13, 14, 22, 23, and 34)

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E., 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105. (cited on pages 2 and 12)

LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P.; et al., 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86, 11 (1998), 2278–2324. (cited on pages 2 and 12)

Li, M.; Zhang, T.; Chen, Y.; and Smola, A. J., 2014. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14 (New York, New York, USA, 2014), 661–670. ACM, New York, NY, USA. doi:10.1145/2623330.2623612. http://doi.acm.org/10.1145/2623330.2623612. (cited on page 51)

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R., 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, (2015). (cited on page 14)

Li, Z.; Chen, Q.; and Koltun, V., 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, 539–548. (cited on page 16)

Long, D. and Fox, M., 1999. Efficient implementation of the plan graph in stan. *Journal of Artificial Intelligence Research*, 10 (1999), 87–115. (cited on page 67)

Long, D. and Fox, M., 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20 (2003), 1–59. (cited on page 68)

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 (2000), 13–13. (cited on page 65)

Luks, E. M., 1999. Hypergraph isomorphism and structural equivalence of boolean functions. In *STOC*. Citeseer. (cited on page 59)

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M., 2019. Online planner selection with graph neural networks and adaptive scheduling. *arXiv preprint arXiv:1811.00210*, (2019). (cited on page 15)

Maas, A. L.; Hannun, A. Y.; and Ng, A. Y., 2013. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning (ICML)*. (cited on page 55)

Mausam and Kolobov, A., 2012. *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool Publishers. ISBN 1608458865, 9781608458868. (cited on page 5)

Nair, V. and Hinton, G. E., 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, 807–814. (cited on pages 11 and 22)

Pearl, J., 1984. Heuristics: intelligent search strategies for computer problem solving. (1984). (cited on page 54)

Pierrot, T.; Ligner, G.; Reed, S. E.; Sigaud, O.; Perrin, N.; Laterre, A.; Kas, D.; Beguir, K.; and de Freitas, N., 2019. Learning compositional neural programs with recursive tree search and planning. In *Conference on Neural Information Processing Systems*. (cited on page 66)

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J., 2015. From non-negative to general operator cost partitioning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*. (cited on page 9)

Ratner, D. and Warmuth, M., 1986. Finding a shortest solution for the nxn extension of the 15-puzzle is intractable. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI'86 (Philadelphia, Pennsylvania, 1986), 168–172. AAAI Press. http://dl.acm.org/citation.cfm?id=2887770.2887797. (cited on page 68)

Richter, S. and Westphal, M., 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39 (2010), 127–177. (cited on page 42)

Ruml, W.; Do, M. B.; Zhou, R.; and Fromherz, M. P., 2011. On-line planning and scheduling: An application to controlling modular printers. *Journal of Artificial Intelligence Research*, 40 (2011), 415–468. (cited on page 1)

Russell, S. J. and Norvig, P., 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,. (cited on page 7)

Sanner, S., 2011. Relational dynamic influence diagram language (rddl): Language description. Technical report, NICTA and the Australian National University. (cited on page 16)

Shen, W.; Trevizan, F.; Toyer, S.; Thiébaux, S.; and Xie, L., 2019. Guiding Search with Generalized Policies for Probabilistic Planning. In *Proc. of 12th Annual Symp. on Combinatorial Search (SoCS)*. (cited on pages 3, 47, and 86)

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P., 2019. Deep learning for cost-optimal planning: Task-dependent planner selection. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*. (cited on pages 2 and 15)

Slaney, J. and Thiébaux, S., 2001. Blocks world revisited. *Artificial Intelligence*, 125, 1-2 (2001), 119–153. (cited on page 64)

Steinmetz, M. and Torralba, A., 2019. Bridging the gap between abstractions and critical-path heuristics via hypergraphs. In *International Conference on Automated Planning and Scheduling*. (cited on pages 41 and 92)

Toyer, S., 2017. Generalised policies for probabilistic planning with deep learning. (cited on page 10)

Toyer, S.; Trevizan, F. W.; Thiébaux, S.; and Xie, L., 2019. Asnets: Deep learning for generalised planning. *CoRR*, abs/1908.01362 (2019). http://arxiv.org/abs/1908.01362. (cited on pages 2, 16, 45, 47, 86, and 88)

Trevizan, F.; Thiébaux, S.; and Haslum, P., 2017. Occupation measure heuristics for probabilistic planning. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*. (cited on page 92)

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y., 2018. Graph Attention Networks. *International Conference on Learning Representations*, (2018). https://openreview.net/forum?id=rJXMpikCZ. (cited on page 16)

Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; and Yu, P. S., 2019. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, (2019). (cited on pages 12, 13, 14, and 23)

Xu, F.; He, F.; Xie, E.; and Li, L., 2018. Fast obdd reordering using neural message passing on hypergraph. *arXiv preprint arXiv:1811.02178*, (2018). (cited on page 28)

Yadati, N.; Nimishakavi, M.; Yadav, P.; Louis, A.; and Talukdar, P., 2018. Hypergcn: Hypergraph convolutional networks for semi-supervised classification. *CoRR*, abs/1809.02589 (2018). http://arxiv.org/abs/1809.02589. (cited on pages 19, 23, 24, 25, 35, and 95)

Yadati, N.; Nitin, V.; Nimishakavi, M.; Yadav, P.; Louis, A.; and Talukdar, P., 2019. Link prediction in hypergraphs using graph convolutional networks. https://openreview.net/forum?id=ryeaZhRqFm. (cited on page 28)

Yoon, S.; Fern, A.; and Givan, R., 2008. Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, 9, Apr (2008), 683–718. (cited on pages 16, 17, and 86)

Younes, H. L. and Littman, M. L., 2004. Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. (2004). (cited on pages 2 and 16)

Young, T.; Hazarika, D.; Poria, S.; and Cambria, E., 2018. Recent trends in deep learning based natural language processing. *IEEE Computational Intelligence magazine*, 13, 3 (2018), 55–75. (cited on page 2)

Zaheer, M.; Kottur, S.; Ravanbakhsh, S.; Poczos, B.; Salakhutdinov, R. R.; and Smola, A. J., 2017. Deep sets. In *Advances in neural information processing systems*, 3391–3401. (cited on pages 59 and 91)

Zhou, D.; Huang, J.; and Schölkopf, B., 2007. Learning with hypergraphs: Clustering, classification, and embedding. In *Advances in neural information processing systems*, 1601–1608. (cited on page 28)