**The Australian National University**
2600 ACT | Canberra | Australia


Australian National University

**School of Computing**

College of Engineering and Computer Science (CECS)

# GOOSE: Learning Heuristics and Parallelising Search for Grounded and Lifted Planning

— 24 pt Honours project (S2/S1 2022–2023)

A thesis submitted for the degree
*Bachelor of Science (Advanced) (Honours)*

**By**:
Dillon Z. Chen

**Supervisors:**
Felipe Trevizan
Sylvie Thiébaux

May 2023

## Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

May, Dillon Z. Chen

# Acknowledgements

I would like to acknowledge Felipe Trevizan and Sylvie Thiébaux for their wonderful supervision over the last three semesters. They have provided me with all sorts of remarkable support and guidance, including but not limited to the wealth of time and integrity spent on providing feedback for multiple drafts, boundless freedom in my projects balanced by insightful pointers, and funding for attending research conferences abroad. All of these factors have been instrumental to the success of this project and my growth as a researcher. I am also grateful for the bountiful amount of computing resources Felipe and Sylvie have provided me for this project, including access to the Gadi supercomputer, the planning group computing cluster, and the School of Computing cluster.

I also acknowledge my previous supervisors Pascal Bercher and Qing Wang whose supervision and desire for my success was also indispensable over the course of my research journey. Lastly, I am indebted to my family's continuous support and encouragement throughout my academic pursuits and personal passions.

# Abstract

Artificial intelligence can be categorised into two main paradigms: model-free *learners* and model-based *solvers*. Learners aim to learn functions with specified domains and targets from data and have been popularised by major advancements in deep learning architectures and hardware. They are able to make quick decisions in various tasks such as computer vision and natural language processing and are able to handle noisy data well. However, they struggle at long range reasoning and lack theoretical guarantees for critical tasks. Solvers on the other hand aim to solve problems modelled by a planning expert which require long range reasoning with theoretical guarantees. The reasoning capabilities of solvers come at the expense of computational complexity and difficulty of leveraging parallelism for hardware such as GPUs. In this thesis, we focus on a class of solvers in the form of planners, which 'plan' by finding a course of actions to taken to reach a specified goal.

This thesis combines the best of both worlds by taking advantage of the capabilities of learners to speedup planners for solving large scale reasoning problems. We do so by introducing our **G**raph neural networks **O**ptimised f**Or** **S**earch **E**valuation (**GOOSE**) framework for learning heuristic functions for guiding search during planning. The two learning tasks we focus on are learning domain-dependent heuristic functions from small problems of a given planning domain for use in much larger problems from the same domain, and learning domain-independent heuristic functions, a form of zero-shot learning where we learn heuristic functions from a set of domains for use in problems from unseen domains.

Our contributions can be categorised into four main themes. We **model** and construct various, novel graph representations of both grounded and lifted planning tasks for use to learn heuristics. The construction of such graphs are complemented with **theory** which aim to answer the question *what domain-independent heuristics can we learn?* On the planning side of our work, we introduce efficient **parallelisation** techniques for speeding up heuristic search using learned heuristic functions for planning. Our final contribution consists of combining all our previous components into GOOSE and evaluating it with a new and comprehensive set of experiments which sets a new standard for the field of **learning for planning**.

# Table of Contents

# List of Tables

# List of Figures

Chapter 1

# Introduction

## 1.1 AI, deep learning, and planning

The concept of artificial intelligence (AI) was introduced by Turing through the Turing test with the prompt *'Can machines think?'* One can further trace back the roots of AI to Alan Turing's universal computer [Turing, 1937], also known as the Turing machine, as a second proof to Hilbert and Ackermann's Entscheidungsproblem conjecture. AI up to the 1980s was viewed more as fields of programming, computational complexity and knowledge representation.

Fast forward to the after the 80s and now the 21st century and the view of AI has shifted into two paradigms as described by Hector Geffner [2018]: *learners* which 'learn' model-free functions with specified domains and targets from data, and *solvers* which aim to 'solve' problems modelled by a planning expert with robustness and theoretical guarantees. Learners have been achieving an incredible rate of advancement in both research and industry, starting from the resurgence of neural networks since the advent of ImageNet for image classification [Krizhevsky et al., 2012], advancement in computing power and economics for deep learning [Amodei and Hernandez, 2018], and available data on the web. In the previous year, we have seen major advances in generative models such as large language models (LLMs) for seemingly interactive artificial intelligence. However, learners have a plethora of shortcomings in their current form, as there is no transparency on why they work, they have few theoretical guarantees and are not robust to unseen tasks, and they have many issues concerning fairness and bias. Nevertheless, these are some problems that researchers and engineers in these fields are currently trying to tackle.

Solvers, or specifically *planners* as we will consider in this work, reason about the given model in order to 'plan' with the objective of computing sequences of actions to solve the current task at hand. They are robust and general in the sense that they are

guaranteed to solve any instance of their intended task with perfect accuracy given enough time and memory. The caveat as stated in the previous sentence is that planning problems are generally intractable to solve. Furthermore, there is no one state-of-the-art planner [Helmert and Domshlak, 2009, Seipp et al., 2020, Richter and Westphal, 2010] which is tractable for all instances [Wolpert and Macready, 1995]. The difficulty arises from the construction of the planners' models which are often complex and large in order to effectively model the problems we would like to solve. Deciding what aspects of the world we want to model, such as uncertainty in the world and the degree of observability of our agents, coupled with the curse of dimensionality makes planning a difficult task of balancing the expressivity and representability. We want our models to be applicable in the real world, but also they have to be solvable with respect to our computing resources. Thus, it is no surprise that mainstream learners, which mimic information from data in order to perform specified one step tasks, are not able to match the reasoning capabilities of planners [Valmeekam et al., 2022, 2023].

## 1.2 Learning for planning, planning for learning, and why we need both

Thus, it is a reasonable idea to combine the two branches of AI to compensate for each other's weaknesses. The main method we explore is by leveraging the immense advancement of deep learning hardware and techniques to alleviate the intractability of sequential planning. As we discussed, the main weakness of planning is their inherent computational complexity. Propositional planning, the simplest form of planning, is PSPACE-complete [Bylander, 1994]. By considering further extensions of planning such as incorporating uncertainty, more compact formalisms and features to model numerical values and time, the complexity of planning can become much more difficult. It is also the case that there is still no convincing algorithm which utilises parallelisation to speed up domain-independent planning, given that it is accepted that we can no longer expect exponential advancements in sequential hardware. Thus, researchers are still relying on studying and improving algorithms rather than advances in hardware in order to speed up planning. ASNets [Toyer et al., 2018] was a wake up call for the field of learning for planning. It displayed convincing experimental results on certain planning domains with respect to classical state-of-the-art algorithms for planning. However, since ASNets' inception, there has not been any other convincing learning architecture that can generalise to solving difficult planning problems outside its training set quickly, with experiments in the literature limited to solving small problems. This thesis focuses on developing learners for planners that are competitive on very large problems and are also optimised for speed.

Although we do not study the topic of planning for learning in this thesis, we also mention that planning and ideas from planning can conversely help learning. For example, the main limitations of model free reinforcement learning (RL) are sample inefficiency, where RL algorithms require a large number of interactions with the world in order to learn

anything useful, and the difficulty of designing useful reward functions. Planning and symbolic action models can be employed with RL techniques [Illanes et al., 2020] in order to inject some information about the environment such that the learner doesn't have to learn the environment from scratch. In computer vision, it is often the case that we implicitly model the physics and invariant structures of the world into the architectures. It has also been suggested that we should build artificial intelligence with insights from cognitive sciences, stating that machines should be able to learn to model the world [Lake et al., 2016].

Robots can be seen as the closest tangible architecture to artificial intelligence which combines both learning and planning: where learning is used for sensing information about the world and planning is used to reason what to do with our available information.

## 1.3 Contributions

This work focuses on constructing fast models which use learning for planning as well as providing formal theoretical insights into such work and other methods in the literature. At the core of the thesis is our proposed **G**raph neural networks **O**ptimised f**O**r **S**earch **E**valuation (**GOOSE**) framework for learning heuristic functions and solving planning tasks quickly. We categorise our contributions into four main themes:

### Modelling

We construct improved, novel graph representations of planning tasks for the goal of learning for planning by combining graph representation learning techniques such as graph neural networks. The graphs are also defined with domain-independent learning in mind, meaning that we can train on data from specified planning domains and perform inference on unseen planning domains. This work is the first to do so without any required assumptions on the planning domains we work with.

### Theory

On top of defining novel graphs, we provide a comprehensive theoretical analysis of what we can and can not learn with our GOOSE framework. We identify classes of domain-independent heuristics GOOSE is able to learn with different graph representations. The theoretical results are also complemented by a set of experiments for answering open questions.

### Parallelisation

On the planning side of our GOOSE framework, we develop intelligent algorithms for optimising the runtime of our learned neural network heuristic functions for search with effective GPU utilisation. This is also a novel contribution that goes beyond naive algorithms of the flavour of batching as many heuristic evaluations during search which we identify as a suboptimal technique.

**Learning for planning**

Our last contribution includes combining all the components of GOOSE for the goal of learning to solve planning problems quickly. We propose a new standard of rigorous and comprehensive experiments for the field of learning for planning with which we use to evaluate GOOSE in comparison to classical planning techniques. This in turn allows us to transparently identify both strengths and limitations of this work as well as provide additional insights to our theoretical results.

## 1.4 Structure of the thesis

In order to present our contributions, we have structured our thesis in the following way:

- Chapter 2 provides the core concepts of planning and graph representation learning used in this work. On the planning side, we motivate and formalise various planning representations and discuss heuristic search, the current state-of-the-art for solving the class of planning problems we are interested in. On the learning side, we focus on graph representation learning with the assumption that readers are already familiar with mainstream deep learning concepts such as neural networks and how they can be trained.
- Chapter 3 presents our first set of novel contributions, namely by formally defining novel graph representations of planning tasks with domain-independent learning in mind. They are carefully constructed by building on planning formalisms described in the background chapter.
- Chapter 4 presents our second set of contributions with a comprehensive theoretical analysis of what we can and can not learn with our graphs in conjunction with graph representation learning methods. We also identify further optimistic directions of research to account for theoretical limits we have identified.
- Chapter 5 encapsulates our first set of experiments to support our theoretical results and complement open questions and discussions from our theory chapter. This can be seen as the set of experiments focused purely on evaluating the learning side of this work with learning metrics focused on the target function being learned.
- Chapter 6 introduces the GOOSE framework and also provides our contributions to the planning side of this work. We develop more intelligent algorithms we use in order to make our learned neural network heuristic functions feasible and competitive for practical use.
- Chapter 7 presents the main experimental setup and results which align with the goal of our thesis: to construct an efficient and practical architecture which leverages learning for planning. We explicitly introduce our GOOSE framework by combining all the components mentioned in the previous chapters. Then we evaluate GOOSE on a diverse set of difficult benchmarks which require our learning

component to generalise from training samples constructed from small planning instances in order to tackle the curse of dimensionality associated with solving large problems. These experiments evaluate the effectiveness of the learned heuristic functions during search, as opposed to studying the learning error and other metrics employed in the first set of experiments.

- Chapter 8 surveys related work in the field of learning for planning, and illustrates where this work places in the current literature.
- Chapter 9 concludes this work by discussing our main contributions and results, and identifying promising future work and the dominant open questions and challenges of the field.

Chapter 2

# Background

In this section, we provide the necessary formal definitions required for our work as well as additional background and historical notes describing the major directions of research and state-of-the-art of the respective fields. The three main components we explore are planning formalisms for representing planning tasks in Sec. 2.1, heuristics and heuristic search for solving planning problems in Sec 2.2, and graph neural networks for performing inference tasks on graph structured data in Sec. 2.3.

## 2.1 Planning formalisms

Here we introduce different formalisms used for representing planning problems. For our work, we further restrict our focus to the classical setting in which the world is deterministic, fully observable and states are discrete [Geffner and Bonet, 2013]. The classical planning model can be explicitly described by its state space, where a state represents the world at a given point and actions provide us a method of transitioning between states.

**Definition 1** (Planning task)**.** A planning task is a state model $\Pi = \langle S, A, s_0, G \rangle$ where $S$ is a set of states, $A$ is a set of actions, where each action $a \in A$ is a function $a : S \to S \cup \bot$ with $a(s) = \bot$ if the action is not applicable in $s$, or otherwise $a(s) = s' \in S$ which is the successor of a state when applying action $a$ in $s$, and has an associated cost $c(a) \in \mathbb{N}$, an initial state $s_0$, and a set of goal sets $G$.

A solution or a *plan* for a planning task is a sequence of actions $\pi = a_1, \ldots, a_n$ where we define $s_i$ as the successor state of applying $a_i$ in $s_{i-1}$ and we have $s_n \in G$. In other words, a plan is a sequence of valid actions which when executed progresses our initial state to a goal state. The cost of a plan $\pi$ is given by $c(\pi) = \sum_{i=1}^{n} c(a_i)$. A planning task is *solvable* if there exists at least one plan. ■

In practice, nobody encodes the state space of each planning task explicitly as the size of the state space of real life problems are generally too large to enumerate and write in memory. Thus the planning community has constructed various planning formalisms each with their advantages and disadvantages. In this section we will focus on three of the main formalisms for pedagogical reasons.

### 2.1.1 Propositional STRIPS

The simplest representation of a planning task is given in propositional STRIPS in which states are modelled as a subset of binary facts. Then we can compactly represent the state space by representing each state as a subset of facts that hold true in the given state.

**Definition 2** (STRIPS planning task)**.** A *propositional STRIPS planning task* is a tuple $\Pi = \langle P, A, s_0, G \rangle$ with $P$ a set of propositions (or facts), $A$ a set of actions, $s_0 \subseteq P$ an initial state and $G \subseteq P$ the goal condition. A state $s$ is a subset of $P$ and is a goal state if $G \subseteq s$.

An action $a \in A$ is a tuple $\langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$ with $\text{pre}(a), \text{add}(a), \text{del}(a) \subseteq P$ and $\text{add}(a) \cap \text{del}(a) = \emptyset$ and has an associated cost $c(a) \in \mathbb{N}$. An action $a$ is applicable in a state $s$ if $\text{pre}(a) \subseteq s$ in which case applying $a$ in $s$ results in the successor state $s' = (s \setminus \text{del}(a)) \cup \text{add}(a)$. ■

The *delete relaxation* of a STRIPS problem $\Pi$ ignores the delete effects on actions and is defined to be $\Pi^+ = \langle P, A^+, s_0, G \rangle$ with $A^+ = \{\langle \text{pre}(a), \text{add}(a), \emptyset \rangle \mid a \in A\}$.

The main advantage of the propositional STRIPS formalism is its simplicity. However, this simplicity comes with several drawbacks such as large representation sizes when trying to model certain problems. As a running example, let us consider the canonical Blocksworld domain. The Blocksworld domain consists of a set of $n$ unique blocks labelled as integers and a table, where blocks are stacked on each other or sitting on the table. The goal of the Blocksworld domain is to restack the blocks in order to construct a certain configuration of towers. Fig. 2.1 illustrates a Blocksworld instance specified by its initial state and a goal state.

To model a state of the Blocksworld problem in propositional STRIPS, we have at least $n^2$ propositions `(on a b)` with $\texttt{a}, \texttt{b} \in [n] := \{1, \ldots, n\}$ representing that block `a` is on block `b`, alongside various other propositions. This allows us to represent how blocks on stacked in the world. However, one may notice for any state that for each `a`, there is at most one `b` such that `(on a b)` is true. In other words we have many pairs of mutually exclusive propositions.

### 2.1.2 Finite Domain Representation

One can reduce the number of location propositions by looking at more compact representations of problems such as the Finite Domain Representation (FDR) [Helmert, 2009]

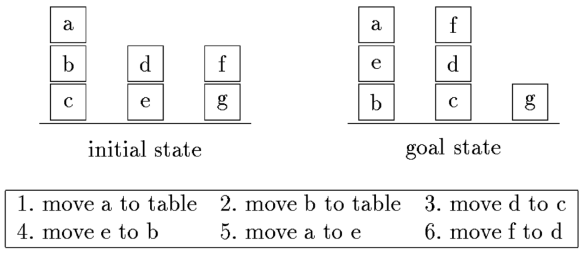



Figure 2.1: A Blocksworld instance with a description of a plan, from [Slaney and Thiébaux, 2001]. This figure is seen again in Ch. 7.

where now a state may be represented by a set of variables, where each variable has its own finite domain. It originated from the SAS+ formalism [Bäckström and Nebel, 1995] where the main difference between the two formalisms lies in the existence of prevail conditions in SAS+ which may be compiled away into FDR. The SAS in SAS+ stands for *Simplified Action Structures* [Bäckström and Klein, 1991].

**Definition 3** (FDR/SAS+ planning task)**.** An *FDR (or SAS+) planning task* is a tuple $\Pi = \langle \mathcal{V}, A, s_0, s_\star \rangle$ where $\mathcal{V}$ is a finite set of state variables $v$, each with a finite domain $D_v$. A fact is a pair $\langle v, d \rangle$ where $v \in \mathcal{V}, d \in D_v$. A partial variable assignment is a set of facts where each variable appears at most once. A total variable assignment is a partial variable assignment where each variable appears at least once. The initial state $s_0$ is a total variable assignment and the goal condition $s_\star$ is a partial variable assignment.

Again, $A$ is a set of actions $a$ of the form $a = \langle \text{pre}(a), \text{eff}(a) \rangle$ where $\text{pre}(a)$ and $\text{eff}(a)$ are partial variable assignments. An action $a$ is applicable in $s$ if $\text{pre}(a) \subseteq s$ in which case applying $a$ in $s$ gives us the successor state $s' = (s \cup \text{eff}(a)) \setminus \{\langle v, d \rangle \in s \mid \exists d' \in D_v, \langle v, d' \rangle \in \text{eff}(a) \wedge d \neq d'\}$. ■

The *delete relaxation* of an FDR planning task $\Pi$ redefines the successor state of applying $a$ in $s$ by $s' = (s \cup \text{eff}(a))$ and allows variables to appear more than once in a state.

Returning to our Blocksworld example, we may represent the configuration of blocks by variables `on_a` each with the corresponding domain $\{\texttt{table}\} \cup [n] \setminus \{\texttt{a}\}$ as each block can be on top of another block or on the table. We note that there exist automatic methods for converting propositional planning tasks into FDR representation [Helmert, 2009]. Aside from encoding mutexes, FDR representations are crucial for constructing abstraction heuristics such as pattern database heuristics which solves projections of problems onto subsets of variables [Edelkamp, 2001, Haslum et al., 2007].

### 2.1.3 Lifted STRIPS

One issue with the previous representations is that their input size usually grows exponentially with the number of objects depending on the domain of choice. This is a result of instantiating all possible combinations of propositional facts and actions. For example in the Blocksworld domain, we have $O(n^2)$ actions for both propositional STRIPS and

FDR formalisms to represent all possible actions for the `stack` and `unstack` actions for stacking/unstacking a block `a` on top of another block `b`. In other words, these two actions have arity 2 as they take two possible arguments and thus representing them all explicitly means there are $O(n^2)$ copies of them. There exist large domains [Masoumi et al., 2015, Gnad et al., 2019, Lauer et al., 2021] where we have actions with high arity $k$ or a large number of objects $n$ such that $O(n^k)$ is very large. This usually means that we are not even able to represent the problem in memory.

This motivates a lifted representation which only encodes the first order information of propositions and actions via predicates and action schema. In fact, this is usually how planning tasks are encoded as input into planners in the PDDL language. We follow the notation of [Lauer et al., 2021] for the definition of lifted STRIPS planning tasks.

**Definition 4** (Lifted STRIPS planning task)**.** A *lifted (STRIPS) planning task* is a tuple $\Pi = \langle \mathcal{P}, \mathcal{O}, \mathcal{A}, s_0, G \rangle$ where $\mathcal{P}$ is a set of first-order predicates, $\mathcal{A}$ is a set of action schema, $\mathcal{O}$ is a set of objects, $s_0$ is the initial state and $G$ is the goal condition. A predicate $P \in \mathcal{P}$ has a tuple of parameters[1] $P(x_1, \ldots, x_{n_P})$ for $n_P \in \mathbb{N}$, noting that $n_P$ depends on $P$ and it is possible for a predicate to have no parameters. We say that a predicate with $n$ parameters is an $n$-ary predicate. A predicate can be instantiated by assigning some of the $x_i$ with objects from $\mathcal{O}$ or other defined variables. A predicate where all variables are assigned with objects is grounded, and is known as a ground proposition. The initial state and goal condition are sets of ground propositions.

An action schema $a \in \mathcal{A}$ is a tuple $\langle \Delta(a), \mathrm{pre}(a), \mathrm{add}(a), \mathrm{del}(a) \rangle$ where $\Delta(a)$ is a set of parameter variables and $\mathrm{pre}(a), \mathrm{add}(a)$ and $\mathrm{del}(a)$ are sets of predicates from $\mathcal{P}$ instantiated with either parameter variables or objects in $\Delta(a) \cup \mathcal{O}$. Similarly to predicates, an action schema with $n = |\Delta(a)|$ parameter variables is an $n$-ary action schema. ■

An action schema where each variable $y \in \Delta(a)$ is instantiated with an object is known as a ground action, or just action. A lifted STRIPS planning task can induce a propositional STRIPS planning task by grounding all predicates and actions. The definition of action application on states for lifted planning tasks is the same as in the propositional case.

Because of the more compact representation of lifted planning tasks, we gain a more expressive formalism. Indeed, solving lifted planning tasks is known to be EXPTIME-complete [Erol et al., 1995]. This can be compared to the propositional case which is PSPACE-complete [Bylander, 1994] to solve. Furthermore, it is known from the polynomial time hierarchy [Stockmeyer, 1976] that EXPTIME is strictly harder than PSPACE. This is in contrast to how it is unknown whether P=NP and/or NP=PSPACE.

### 2.1.4 Additional comments

*Satisficing planning* refers to determining whether a planning task is solvable. On the other hand, *optimal planning* refers to finding an optimal plan, i.e. a plan with minimal cost over all plans, for a given planning task which is assumed to be solvable. Both

[1] One can further define types associated to each variable.

variants are known to be PSPACE-complete in the propositional case [Bylander, 1994] by reducing from polynomial space bounded Turing machines. However in practice, optimal planning is a much more difficult problem. Satisficing planning for delete relaxed problems is in polynomial time by simply greedily applying actions until we either cannot do so anymore or until the goal is reached, noting that plans are bounded by the size of the number of actions. On the other hand, optimal planning for delete relaxed problems is NP-complete [Bylander, 1994] by reducing from 3SAT. However, we do not know whether $\mathsf{P} = \mathsf{NP}$ or $\mathsf{NP} = \mathsf{PSPACE}$ yet. Complexity analysis is a good baseline tool for measuring the expressiveness of planning formalisms and has been done for all sorts of planning formalism variants and extensions such as for probabilistic planning [Littman, 1997] and hierarchical task network (HTN) planning and its nondeterministic extensions [Erol et al., 1996, Alford et al., 2015, Chen and Bercher, 2021, 2022].

The canonical method for constructing planning tasks as inputs into planners is through the Planning Domain Definition Language (PDDL). After parsing a PDDL file, planners usually translate the task into a lifted planning task, followed by an optional step of grounding it into a propositional STRIPS planning problem and finally converting it into an FDR planning task. There exist various versions and extensions to model more expressive domains and features such as making use of conditional and quantified conditions and effects, state dependent axioms, numeric variables, and time in temporal planning. We refer the interested reader to [Haslum et al., 2019] for more details on PDDL and what features it can handle.

## 2.2 Heuristic search

Heuristic search is a powerful and the de-facto method for solving planning problems, resulting in the common conception that 'planning is just search'. Heuristic search methods generally consist of two core components:

1. a heuristic (value) function, which gives an estimate of the optimal cost to the goal from any given state, and
2. a heuristic search algorithm which uses a heuristic to guide search and reduce the number of expansions, hence saving time and memory.

There also exist many complementary methods for improving heuristic search such as computing mutexes [Helmert, 2006, Alcázar and Torralba, 2015] and symmetries [Pochter et al., 2011, Abdulaziz et al., 2015, Sievers et al., 2019] to restrict and simplify the search space through pruning unnecessary states. An important direction of research in heuristic search is representation of the problem. More specifically, we would like to work with a representation that is polynomial in the input PDDL [Lauer et al., 2021] such as the lifted formalism described previously instead of a grounded representation which may occur an exponential blowup in the input. However, working in the lifted representation is not a trivial task. In the only known lifted planner [Corrêa et al., 2020], the generation of successor states is done via conjunctive database queries which is in

general an NP-hard problem. Nevertheless, this does not rule out the possibility that there may exist a polynomial time successor state generator.

A heuristic value function or simply *heuristic* for a problem with state space $S$ has the form $h : S \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. An $\infty$ value may be assigned to *dead ends*, states in which there exist plan. The optimal heuristic $h^*$ assigns each state $s$ the cost of the optimal plan of reaching the goal from that state if such a plan exists, else it assigns $\infty$. An heuristic is *safe* if $h(s) = \infty$ implies $h^*(s) = \infty$ for all $s \in S$. An heuristic $h$ is *admissible* if $h(s) \leq h^*(s)$ for all $s \in S$. An heuristic is *goal aware* if $h(g) = 0$ for all goal states $g$. An heuristic is *consistent* if it is goal aware and $h(s) \leq c(a) + h(s')$ for all $s \in S$, $a \in A$ and $s' = a(s)$. An heuristic which is consistent is admissible although the converse is not true. A heuristic $h$ *dominates* another heuristic $h'$ if $h(s) \geq h'(s)$ for all $s \in S$. Intuitively, higher admissible heuristic value costs result in better performance in A$^*$ search although this is not a theoretical guarantee [Holte, 2010]. We refer to Sec. 2.2.2 for specific examples of constructing heuristic functions.

### 2.2.1 Heuristic search algorithms

Heuristic search algorithms utilise a heuristic to decide the order of nodes to expand during a graph search of the transition graph induced by a planning problem, where a node holds information about a state in the planning task and also the current partial plan which takes us to that state from the initial state. The frontier of a heuristic search algorithm is generally represented by a priority queue and referred to as the *open* list, and any node that is not currently in the queue but has been in the queue before is in the *closed* list.

The A$^*$ algorithm is a well known example of a heuristic search algorithm with optimality and other theoretical guarantees [Pearl, 1984]. Given a search problem, let $g$ be the function which assigns a node $n$ the cost of the partial plan stored in the node. Our heuristic function $h$ can evaluate nodes by evaluating the corresponding state. Then each iteration of A$^*$ expands a node $n$ in the search frontier with the lowest $f(n) = g(n)+h(n)$ value. Plans found by A$^*$ are optimal when $h$ is admissible. Consistent heuristics ensure that A$^*$ does not have to re-expand nodes.

Another common heuristic search algorithm is Greedy Best First Search (GBFS) which pops nodes from the frontier in order of their lowest $f(n) = h(n)$ value. GBFS does not have any optimality guarantees but as a tradeoff is generally more efficient for satisficing planning.

Besides these two canonical heuristic search algorithms, there also exist a whole zoo of improved or alternate heuristic search algorithms. Iterative Deepening A$^*$ (IDA$^*$) [Korf, 1985] and Recursive Best First Search (RBFS) [Korf, 1993] can be seen as DFS versions of A$^*$ and GBFS respectively, which sacrifice solving time for linear space complexity. Performing DFS lookaheads for GBFS [Stern et al., 2010] and A$^*$ [Bu et al., 2014] can reduce the time complexity overhead of purely DFS methods. Best-First Width Search [Lipovetzky and Geffner, 2017] combines the advantages of the exploitation as-

pects of heuristic search for guiding search towards the goal and the exploration aspects of Width-Based Search [Lipovetzky and Geffner, 2012] which aids in avoiding local minima or plateaus in the search space by attempting to guide the planner to explore states which look different to those that have been explored before with a novelty measure. Local search methods can also be integrated into GBFS [Xie et al., 2014] as forms of exploration to deal with such local minima and plateaus.

Bidirectional A* [Pohl, 1969] performs two A* search algorithms simultaneously, a forward search from the initial state and a backwards search from a specified goal node. Its advantages are outweighed by its difficult termination criteria for proving optimality and the need to specify a goal state for planning problems. Symbolic Bidirectional A* [Torralba et al., 2016] combines heuristic search with abstraction heuristics and symbolic search [McMillan, 1993] which reasons over sets of states in order to improve the termination criterion of bidirectional search.

All the methods described so far are *eager* search methods which evaluate the heuristic function of a state the moment it is generated. However, there also exists *lazy* search [Helmert, 2006, Richter and Helmert, 2009] in which the heuristic evaluation of a node is delayed until it gets expanded from the search queue. In turn, a node's value for the priority queue in search is determined by its parent's heuristic value. This *deferred evaluation* of heuristic values sometimes provide a performance boost in solving speed in cases when there are many more generated nodes than expanded and evaluated nodes. For reference, we provide the pseudocode of eager and lazy GBFS in Alg. 1 and 2 without node reopenings.

*Preferred operators* [Hoffmann and Nebel, 2001, Helmert, 2006, Richter and Helmert, 2009] are as the name suggests operators which are preferred and useful in some way and are often provided alongside the computation of a heuristic value. In eager search, preferred operators may be used as additional information to guide search by pruning away successor nodes that are not constructed by the preferred operator. They may also be used in lazy search with a dual-queue search where one queue stores nodes generated by preferred operators which results in an additional priority for preferred operators. The combination of preferred operators and lazy search offers a significant performance boost over eager search alone [Richter and Helmert, 2009, Corrêa et al., 2020].

### 2.2.2 Heuristic functions

In this section we survey *domain-independent heuristic functions* since we are focused on automation in planning. This is in opposition to *domain-dependent heuristic functions* which can be seen as handcrafted heuristics for specific planning problems. An example of this is using the Manhattan or L1 distance heuristic for pathfinding on grids.

The baseline heuristic function is the *zero* heuristic $h = 0$ which provides no information about states. However, when used in conjunction with A* search, we usually define the baseline to be $h(s) = 1$ for all non-goal states and $h(s) = 0$ for goal states. The reasoning for this is that A* only checks whether a state satisfies the goal condition

**Algorithm 1:** Eager GBFS

**Data:** Planning problem $\langle S, A, s_0, G \rangle$; heuristic function $h$.

```
1  OPEN ← ∅
2  s.closed ← ⊥,   ∀s ∈ S
3  OPEN.push(s_0, 0)
4  while OPEN ≠ ∅ do
5      s ← OPEN.popFront()
6      s.closed ← ⊤
7      for a ∈ A do
8          if a(s) = ⊥ then
9              continue
10         t ← a(s)
11         if t ∈ G then
12             return Extract plan from t
13         if t.closed = ⊥ then
14             OPEN.push(t, h(t))
15 return No solution
```

when it is popped from the frontier and not when it is generated as a successor. With the zero heuristic ($h = 0$), this results in unnecessary node expansions, and with very bad tie breaking could result in expanding the whole state space. We can constrast this to breadth first search (BrFS) where the goal condition checked when successors are generated which is sound and complete for unitary action costs.

Another simple heuristic is the *goal count* heuristic, $h^{\text{gc}}$, which counts the number of unachieved propositional goals remaining. Despite its simplicity, it performs very well on certain domains due to its computation speed and informedness for simple domains. However, like many of the more informed heuristics we will begin to explore, the goal count heuristic is defined for the previously discussed planning formalisms and not enumerated representations.

The most powerful heuristic is the *perfect heuristic* $h^*$ which is the cost of the optimal plan from any given state. Of course, computing it is infeasible and usually amounts to solving the planning task by itself all together.

### Delete relaxation heuristics

Early work on domain-independent heuristics focused on the delete relaxation of the problem. We begin with the most powerful delete relaxation heuristic: the *perfect delete relaxation heuristic* $h^+$. This is the cost of the optimal delete relax plan from any given state. One method of computing it involves iteratively solving a minimum cost hitting set problem for an increasing set of disjunctive action landmarks [Haslum et al., 2012].

**Algorithm 2:** Lazy GBFS

**Data:** Planning problem $\langle S, A, s_0, G\rangle$; heuristic function $h$.

```
OPEN ← ∅
s.closed ← ⊥,   ∀s ∈ S
OPEN.push(s_0, 0)
while OPEN ≠ ∅ do
    s ← OPEN.popFront()
    s.closed ← ⊤
    s.h ← h(s)
    for a ∈ A do
        if a(s) = ⊥ then
            continue
        t ← a(s)
        if t ∈ G then
            return Extract plan from t
        if t.closed = ⊥ then
            OPEN.push(t, s.h)
return No solution
```

A disjunctive action landmark is itself a set of actions of which at least one contributes to any plan for a delete relaxed problem.

The $h^{\text{add}}$ and $h^{\max}$ heuristics are some of the earliest domain-independent heuristics which are computed on the relaxed problem where we achieve every or the most costly proposition in each subgoal respectively. Their formal definitions are given below and a naive value iteration method for computing them for propositional STRIPS problem are given in Alg. 3 and 4. Their computation and definition for FDR problems can be made in the obvious way by treating variable value pairs as propositions, whereas we refer the reader to [Corrêa et al., 2021] for the computation of $h^{\text{add}}$ and $h^{\max}$ for lifted STRIPS problems without grounding.

**Definition 5** ($h^{\text{add}}/h^{\max}$)**.** Let $\Pi = \langle P, A, s_0, G\rangle$ be a propositional STRIPS problem. The *additive heuristic* $h^{\text{add}}/h^{\max}$ is defined by $h^{\text{add}}(s) = h^{\text{add}}(s, G)$ and $h^{\max}(s) = h^{\max}(s, G)$ where

$$h^{\text{add}}(s, g) = \begin{cases} 0, & \text{if } g \subseteq s \\ \min\limits_{a \in A, p \in \text{add}(a)} [c(a) + h^{\text{add}}(s, \text{pre}(a))], & \text{if } g = \{p\} \\ \sum\limits_{p \in g} h^{\text{add}}(s, \{p\}), & \text{if } |g| > 1. \end{cases} \tag{2.1}$$

and

$$h^{\max}(s,g) = \begin{cases} 0, & \text{if } g \subseteq s \\ \min\limits_{a \in A, p \in \text{add}(a)} [c(a) + h^{\max}(s, \text{pre}(a))], & \text{if } g = \{p\} \\ \max\limits_{p \in g} h^{\max}(s, \{p\}), & \text{if } |g| > 1. \end{cases} \tag{2.2}$$

■

**Algorithm 3:** Naive $h^{\text{add}}$

**Data:** Propositional STRIPS planning task $\Pi = \langle P, A, s_0, G \rangle$
**Result:** $h^{\text{add}}(s) \in \mathbb{N}$

```
1 h^(0)[p] ← 0,   ∀p ∈ s_0
2 h^(0)[p] ← ∞,   ∀p ∈ P \ s_0
3 for i = 1, ... do
4     for a ∈ A do
5         h^(i)[a] ← Σ_{p∈pre(a)} h^(i−1)[p]
6     for p ∈ P do
7         h^(i)[p] ← min(h^(i−1)[p], min_{a∈A, p∈add(a)} h^(i)[a] + c(a))
8     if h^(i) = h^(i−1) then
9         return Σ_{p∈g} h^(i)[p]
```

**Algorithm 4:** Naive $h^{\max}$

**Data:** Propositional STRIPS planning task $\Pi = \langle P, A, s_0, G \rangle$
**Result:** $h^{\max}(s) \in \mathbb{N}$

```
1 h^(0)[p] ← 0,   ∀p ∈ s_0
2 h^(0)[p] ← ∞,   ∀p ∈ P \ s_0
3 for i = 1, ... do
4     for a ∈ A do
5         h^(i)[a] ← max_{p∈pre(a)} h^(i−1)[p]
6     for p ∈ P do
7         h^(i)[p] ← min(h^(i−1)[p], min_{a∈A, p∈add(a)} h^(i)[a] + c(a))
8     if h^(i) = h^(i−1) then
9         return max_{p∈g} h^(i)[p]
```

We note that $h^{\max}$ is an admissible heuristic that is always dominated by $h^+$ and $h^{\text{add}}$ is inadmissible and dominates $h^+$. The $h^{\max}$ algorithm is an optimistic algorithm which assumes that reaching the most costly proposition in a set of propositions is sufficient for achieving all propositions, whereas the $h^{\text{add}}$ algorithm is a pessimistic heuristic which assumes that propositions are always achieved independently of each other.

The $h^{\text{FF}}$ heuristic [Hoffmann and Nebel, 2001] is an approximation of $h^+$ for unitary cost problems. The computation of $h^{\text{FF}}$ can be summarised in two main steps:

1. A progression phase, where we greedily apply actions at our current state whenever possible until we reach our goal condition in the delete free relaxation of the problem.
2. A regression phase, where actions are extracted backwards from the goal state corresponding to a delete relax plan.

This first step is akin to unrolling the delete relaxation planning graph of the Graphplan algorithm [Blum and Furst, 1997], a directed graph for representing a planning task, with alternating layers between proposition and action nodes. The original intent of the planning graph in Graphplan is to compute plans for planning tasks and in the process also stores mutual exclusion relations (mutexes) of pairs of propositions and actions. However, such mutexes can be ignored in the delete relaxation case [Hoffmann and Nebel, 2001]. The $h^{\text{FF}}$ heuristic described previously is not a well defined heuristic as it also requires a tie breaking strategy for selecting actions to apply and extract in both the progression and regression phase. Similarly to $h^{\max}$ and $h^{\text{add}}$, it is possible to compute it for lifted planning without grounding [Corrêa et al., 2022].

The set-additive heuristic $h^{\text{SA}}$ [Keyder and Geffner, 2008] combines the ideas of $h^{\text{add}}$ and $h^{\text{FF}}$ which encodes the cost of a relaxed plan but is also able to deal with arbitrary action costs. It differs from $h^{\text{add}}$ as it aggregates and propagates information with a set union of actions as opposed to the sum of action costs.

### Critical path heuristics

The critical path heuristic $h^m$ can be seen as a generalisation of $h^{\max}$ which instead of computing the best cost of reaching a single goal proposition, computes the best cost of reaching a set of goals of size $m \in \mathbb{N} \setminus \{0\}$. In this way, $h^{\max} = h^1$.

**Definition 6** ($h^m$)**.** Let $\Pi = \langle P, A, s_0, G \rangle$ be a STRIPS problem and $m \in \mathbb{N} \setminus \{0\}$. The *critical path heuristic* $h^m$ is defined by $h^m(s) = h^m(s, G)$ where

$$h^m(s, g) = \begin{cases} 0, & \text{if } g \subseteq s \\ \min\limits_{a \in A, \text{regr}(g,a) \neq \bot} [c(a) + h^m(s, \text{regr}(g, a))], & \text{if } |g| \leq m \\ \max\limits_{g' \subset g, |g'| = m} h^m(s, g'), & \text{if } |g| > m \end{cases} \tag{2.3}$$

and

$$\text{regr}(g, a) = \begin{cases} \bot, & \text{if } \text{add}(a) \cap g = \emptyset \text{ or } \text{del}(a) \cap g \neq \emptyset \\ (g \setminus \text{add}(a)) \cup \text{pre}(a), & \text{else.} \end{cases} \tag{2.4}$$

■

From the definition we can see that $h^m$ takes delete effects into account and thus is not classified as a delete relaxation heuristic. One can also notice that $h^m \leq h^{m+1}$ and furthermore, $h^m = h^*$ for some $m \leq |G| \in \mathbb{N}$. Given a planning problem $P$, the $h^m$ heuristic can be computed by a dynamic programming approach as an extension of Alg. 4 or alternatively by computing $h^{\max}$ on a transformation of a problem [Haslum, 2009]. The computation for $h^m$ is exponential in $m$ which becomes prohibitive for large $m$. In practice, one uses up to $m = 2$. However, it is more common to use the $h^2$ heuristic as a preprocessing step by computing invariants to simplify the planning task [Haslum, 2007, Alcázar and Torralba, 2015].

**More powerful heuristics: abstractions, cost partitioning and more...**

So far we have outlined some baseline heuristics which are still useful today despite their age, namely $h^{\text{add}}$ and $h^{\text{FF}}$ for satisficing planning. However, such heuristics are inadmissible and thus do not guarantee optimal plans when used in conjunction with any heuristic search algorithm such as A$^*$ or GBFS. For the sake of completeness, in this section we briefly survey some powerful admissible heuristics constructed in the previous one and a half decade for *optimal* planning. We will not dwelve into them in much detail as our work focuses primarily on satisficing planning, and it is also known that highly accurate admissible heuristics do not necessarily perform better with satisficing search algorithms such as GBFS [Wilt and Ruml, 2015].

Abstraction heuristics are heuristics which are computed from *abstractions* of our given planning tasks: very informally, simplifications of the problem. One of the earliest abstraction heuristics are pattern databases (PDBs) [Edelkamp, 2001] which are projections of planning problems to subsets of variables, known as a pattern. The heuristic computed from a single pattern may not be informative and thus, one usually considers a combination of patterns as done in the canonical heuristic [Haslum et al., 2007] for a collection of patterns. One requires that patterns are orthogonal in the sense that no actions affect variables in both patterns for them to be combined admissibly. Merge and shrink (M&S) abstractions [Helmert et al., 2007] take a different approach by trying to construct a single good abstraction by searching over the space of abstractions by merging pairs of abstractions or shrinking them. It was shown that M&S abstractions are the most general abstractions in the sense that any abstraction function can be theoretically represented as an M&S abstraction. Furthermore they can compute $h^*$ and even in polynomial time for certain benchmarks. Cartesian abstractions for planning [Seipp and Helmert, 2013], originating from the model-checking community, also generalise PDBs and provide more efficient abstraction refinement techniques than M&S.

Cost partitioning is a powerful method for combining $n$ heuristics in an admissible way by distributing action costs over $n$ copies of a planning problem in an intelligent manner. The method was introduced to planning by computing optimal cost partitionings for heuristics via linear programs [Katz and Domshlak, 2008]. It has also been used to theoretically classify various classes of heuristics through cost partitioning compilations [Helmert and Domshlak, 2009] and produced the LM-cut heuristic as an additive

landmark heuristic dominating $h^{\max}$. Saturated cost partitioning [Seipp et al., 2020] is a greedy method for constructing cost partitionings for a sequence of heuristics by assigning a cost function to the first heuristic in the sequence and allocating the residual cost function for the remaining heuristics in the sequence. Given that the order of the heuristics has an impact on the cost partitioning quality constructed in this way, a set of orders is precomputed with which we take the maximum for each state for evaluation.

### 2.2.3 Brief history of heuristic search and extensions

Heuristic search has been used for solving planning problems since the advent of the first International Planning Competition (IPC) in 1998 and the HSP planner [Bonnet and Geffner, 1998] which combines the $h^{\text{add}}$ heuristic with greedy best first search or hill climbing for satisficing planning. In the following IPC in 2000, the Fast-Forward planning system (FF) [Hoffmann and Nebel, 2001] uses a new heuristic $h^{\text{FF}}$ during search and other heuristic search planners significantly outperformed the SAT and Graphplan based planners. In the 2002 IPC similar conclusions were made with heuristic search providing the best performance. The 2004 IPC introduced optimal planning tracks and although the heuristic search methods performed best in the satisficing track, SAT based planners such as SATPLAN [Kautz et al., 2006] and BlackBox [Kautz and Selman, 1998] won the optimal track. Similar conclusions about heuristic search and planning as SAT was made in the 2006 IPC. The 2008 IPC had updated scoring for optimal planning and the baseline which consisted of A$^*$ with the zero heuristic won. The winning sequential track planner was LAMA [Richter and Westphal, 2010] which combines the $h^{\text{FF}}$ with a heuristic for counting unachieved landmarks. Heuristic search began to dominate in the 2011 IPC and continued to do so in the 2014 and 2018 IPC. We refer to [Torralba and Croitoru, 2019] for a more detailed overview of the history of the IPC. Nevertheless, we do note that there has been a resurgence of planning as SAT as a complementary method to heuristic search with works [Höller and Behnke, 2022] showing where lifted SAT planning outperforms heuristic search.

Heuristic search is not constrained primarily to classical planning and can also be applied to all sorts of planning extensions. It has been applied to deal with probabilistic planning where actions may have multiple sets of effects and associated probabilities. Heuristic search algorithms for these problems include LAO$^*$ [Hansen and Zilberstein, 2001] and LRTDP [Bonet and Geffner, 2003]. There also exist refined heuristic functions which do not simply consider the all outcome determinisation relaxation of the problem [Trevizan et al., 2017b, 2018, Klößner and Hoffmann, 2021]. Heuristic search can also be used on constrained stochastic planning by considering the dual space [Trevizan et al., 2017a]. NAMOA$^*$ [Mandow and Pérez-de-la-Cruz, 2010] is the de-facto heuristic search algorithm for planning with multiple objectives and there exist recent work on heuristic functions which account for the interactions between multiple objectives [Geißer et al., 2022]. Finally we also have heuristic search algorithms MOLAO$^*$ and MOLRTDP and corresponding heuristic functions for planning which combine both probabilities and multiple objectives [Chen et al., 2023].

### 2.2.4 Taxonomy of learning heuristic functions

In this section we had primarily focused on domain-independent algorithms for computing heuristic functions, which we named domain-independent heuristic functions. However, it is more difficult to classify heuristic function learning methods as either domain-dependent or domain-independent. This is because learning *domain-dependent heuristic functions*, heuristic functions which are learned to do well only on problems for a specific domain, can be made an automated process as long as we have an automated method for generating useful training samples and a domain-independent learning algorithm. We name this category of methods for learning domain-dependent heuristics as *domain-independent learning algorithms of domain-dependent heuristics*.

This is in contrast to learning methods which are designed to work for only specific domains. An example of this is learning policies, a function which returns an action to take in any state, for the Sokoban and travelling salesman problem (TSP) domains framed as planning with hand coded translators for problems in each domain [Groshev et al., 2018]. More specifically, the method converts Sokoban states into images for inputs into convolutional neural networks, and TSP states into graphs for inputs into graph neural networks. However, it is possible adapt learning policies to learning heuristic functions. Methods of this category are rare in planning but we name them *domain-dependent learning algorithms of heuristics*.

Lastly, it is possible to learn *domain-independent heuristic functions*, heuristic functions that are learned with the aim of working well on any domain, even on domains unseen in the training data. We name this category of methods for learning domain-independent functions as *learning algorithms of domain-independent heuristics* and are the most general methods as we only have to train a model once for use in any number of domains, whereas the aforementioned methods require training for each domain.

Tab. 2.1 summarises the taxonomies of classical computation and learning of heuristic functions. Note that learning methods with higher levels of generality subsume methods with lower levels of generality. Furthermore, generality is usually inversely proportional to heuristic informedness for search. Our work belongs to the class of *learning algorithms of domain-independent heuristics*. Later in Ch. 8, we provide a comprehensive survey of related work in learning for planning to classify works with the described taxonomy and emphasise the various contributions of our work.

Table 2.1: Levels of generality of different heuristic function algorithm taxonomies.

| | |
|---|---|
| high | domain-independent heuristics |
| | learning algorithms of domain-independent heuristics |
| medium | domain-independent learning algorithms of domain-dependent heuristics |
| low | domain-dependent heuristics |
| | domain-dependent learning algorithms of heuristics |

## 2.3 Graph neural networks

Graph neural networks (GNNs) are a neural network framework which can operate on non-Euclidean data in the form of graphs. Their rise in popularity can be attributed to the concurrent interest on neural networks due to massively increased compute power, and the great wealth of graph theory developed throughout history. Moreover, they are highly applicable given that almost anything can be represented as a graph, with examples ranging from tangible real life applications such as social networks, molecule structures, and knowledge graphs, to abstract applications such as helping in solving combinatorial optimisation and NP-hard problems [Khalil et al., 2017, Li et al., 2018, Cappart et al., 2021]. We will assume that readers are familiar with mainstream deep learning concepts such as neural networks and how they can be trained by optimising a loss function with derivative information which can be computed via backpropagation.

Before looking into GNNs, we first introduce some terminology. In the context of learning tasks, we define a directed graph to be a tuple $\langle V, E, \mathbf{X}, \mathbf{E}\rangle$ where $V$ is a set of nodes, $E \subseteq V \times V$ is a set of edges, $\mathbf{X} : V \to \mathbb{R}^d$ to be a function representing the node features of the graph and $\mathbf{E} : E \to \mathbb{R}^e$ representing the edge features of the graph. Let $n$ be the number of nodes of a graph and $m$ the number of edges. In practice, $V$ is a set of integers, $E$ is represented by a (dense or sparse) matrix, $\mathbf{X}$ is represented by a dense matrix in $\mathbb{R}^{n\times d}$ where $n$ is the number of nodes and $\mathbf{X}[i]$ is the feature of the node $i$, and $\mathbf{E}$ is represented by a matrix in $\mathbb{R}^{m\times e}$ where $\mathbf{E}[i]$ is the feature of the edge with index $i$ in the sparse representation of $E$. A graph is undirected if instead we have $E \subseteq \binom{V}{2}$. A multigraph is the same as a graph except that we may have more than one copy of an edge. To differentiate unique edges with the same endpoint vertices and to make $\mathbf{E}$ well defined, edges in $E$ now have the form $((u, v), a)$ where $a$ is an identifier, or in the case of undirected graphs, $(\{u, v\}, a)$. Furthermore, for graphs without edge features (e.g. if the image of $\mathbf{E}$ is a singleton set), we may simply notate a graph as $\langle V, E, \mathbf{X}\rangle$.

The neighbourhood of a node $u$ in a graph $G$ is given by $\mathcal{N}(u) = \{v \mid (v, u) \in E\}$. Note in the context of directed graphs, this is the set of *in-neighbours*. The reasoning for this is because in implementations of graph neural networks, aggregation steps of neighbourhoods are defined in this way. In other words, information flows in the direction of the arrows. A graph $G' = \langle V', E', \mathbf{X}', \mathbf{E}'\rangle$ is a subgraph of a graph $G = \langle V, E, \mathbf{X}, \mathbf{E}\rangle$ if $V' \subseteq V$, $E' \subseteq (V' \times V') \cap E$, $\mathbf{X}|_{V'} = \mathbf{X}'$ and $\mathbf{E}|_{E'} = \mathbf{E}'$ where $f|_X$ denotes the function $f$ restricted to the domain $X$.

We first introduce graph neural networks by the message passing neural network (MPNN) framework [Gilmer et al., 2017] and later discuss extensions of GNNs in the literature. Note that we do not focus on traditional graph representation learning (GRL) methods such as spectral methods or graph kernels and refer the interested reader to [Hamilton, 2020] for some of these methods.

### 2.3.1 Message passing neural networks

A message passing neural network iteratively updates node embeddings of a graph locally in one-hop neighbourhoods with the general message passing equation

$$\mathbf{h}_u^{(t+1)} = \varphi^{(t)}\left(\mathbf{h}_u^{(t)}, \square_{v\in\mathcal{N}(u)}^{(t)} f^{(t)}\left(\mathbf{h}_u^{(t)}, \mathbf{h}_v^{(t)}, \mathbf{e}_{v,u}^{(t)}\right)\right) \tag{2.5}$$

where in the $t$-th iteration or layer of the network, $\mathbf{h}_u^{(t)} \in \mathbb{R}^{F^{(t)}}$ is the embedding of node $u$, with $\mathbf{h}_u^{(0)} = \mathbf{X}[u]$, and $\mathbf{e}_{v,u}^{(t)} \in \mathbb{R}^{D^{(t)}}$ is the feature of the edge which points from $v$ to $u$. We have that $\varphi^{(t)}$ and $f^{(t)}$ are arbitrary almost everywhere differentiable functions and $\square^{(t)}$ usually a differentiable permutation invariant function acting on sets of vectors such as sum, mean or component wise max.

In the case where we operate on graphs without edge features as will be the case for the majority of our work, the equation can be simplified to be

$$\mathbf{h}_u^{(t+1)} = \varphi^{(t)}\left(\mathbf{h}_u^{(t)}, \square_{v\in\mathcal{N}(u)}^{(t)} f^{(t)}\left(\mathbf{h}_u^{(t)}, \mathbf{h}_v^{(t)}\right)\right). \tag{2.6}$$

In order to satisfy the 'neural network' component of an MPNN, it is common that $\varphi$ or $f$ has learnable parameters, for example $\varphi$ can be given by a feed forward network.

In order for an MPNN to produce a graph representation for an input, it is then common to pool all the node embeddings after a number of message passing updates with a *graph readout* function $\Phi$ which is again usually given by a differentiable permutation invariant function. Alg. 5 outlines the general MPNN framework for graphs which exhibit only node features.

**Algorithm 5:** Message Passing Neural Network

**Data:** Graph $G = (V, E)$, node features $\mathbf{X}$, number of layers $L$, graph readout function $\Phi$, aggregation functions $\square^{(t)}$ and multi-variable functions $\varphi^{(t)}, f^{(t)}$ for $t = 1, \ldots, L$.

**Result:** Graph embedding $\mathbf{h}_G$.

$\mathbf{h}_u^{(0)} \leftarrow \mathbf{X}[u], \quad \forall u \in V$

**for** $t = 0, \ldots, L-1$ **do**

  $\mathbf{h}_u^{(t+1)} = \varphi^{(t)}\left(\mathbf{h}_u^{(t)}, \square_{v\in\mathcal{N}(u)}^{(t)} f^{(t)}\left(\mathbf{h}_u^{(t)}, \mathbf{h}_v^{(t)}\right)\right), \quad \forall u \in V$

$\mathbf{h}_G \leftarrow \Phi_{u\in V}\left(\mathbf{h}_u^{(L)}\right)$

**return** $\mathbf{h}_G$

A canonical example of an MPNN is the Graph Convolutional Network (GCN) [Kipf and Welling, 2017] with update equation given by

$$\mathbf{h}_u^{(t+1)} = \sigma\left(\mathbf{W}^{(t)} \sum_{v\in\mathcal{N}(u)\cup\{u\}} \frac{\mathbf{h}_v^{(t)}}{\sqrt{|\mathcal{N}(u)|\,|\mathcal{N}(v)|}}\right) \tag{2.7}$$

where if $\mathbf{h}_v^{(t)} \in \mathbb{R}^{d_2}$ and $\mathbf{h}_u^{(t+1)} \in \mathbb{R}^{d_1}$, then $\mathbf{W}^{(t)} \in \mathbb{R}^{d_1 \times d_2}$ is a matrix with learnable parameters, and $\sigma : \mathbb{R} \to \mathbb{R}$ is a nonlinear activation function usually given by the sigmoid or ReLU function. The GCN update equation is motivated by graph spectral theory and uses the normalised graph Laplacian to update node embeddings by local averages. GCN performs well on transductive tasks where the graph structure is fixed, for example by predicting hidden node labels using other shown node labels on the same graph as training. However, they only tend to perform well on homogeneous graphs, graphs where neighbouring nodes share similar labels. The class of graph neural networks which are motivated by spectral methods such as GCN and ChebNet [Defferrard et al., 2016] are commonly referred to as *spectral* GNNs.

Another example is the Graph Attention Network (GAT) [Velickovic et al., 2017] which leverages the multi-head attention mechanism [Vaswani et al., 2017] for dealing with variable size inputs and is one of the foundations for the state-of-the-art models for NLP tasks. In the case of graphs, node neighbourhoods are the variable size inputs. The update equation is given by

$$\mathbf{h}_u^{(t+1)} = \Big\Vert_{k=1}^{K} \sigma\Big( \sum_{v \in \mathcal{N}(u)} \alpha_{u,v}^{(k,t)} \mathbf{W}^{(t)} \mathbf{h}_v^{(t)} \Big) \tag{2.8}$$

where as previously, $\mathbf{h}_u^{(t)} \in \mathbb{R}^{d_2}, \mathbf{h}_u^{(t+1)} \in \mathbb{R}^{d_1}, \mathbf{W}^{(t)} \in \mathbb{R}^{d_1 \times d_2}$, $\sigma : \mathbb{R} \to \mathbb{R}$ a nonlinear activation function, $\Vert$ denotes vector concatenation, and $\alpha_{u,v}^{(k,t)} \in \mathbb{R}$ are attention weights given by

$$\alpha_{u,v}^{(k,t)} = \mathrm{softmax}_v(e_{u,v}^{(k,t)}) = \frac{\exp(e_{u,v}^{(k,t)})}{\sum_{w \in \mathcal{N}(u)} \exp(e_{u,w}^{(k,t)})} \tag{2.9}$$

which normalises with the softmax function the attention coefficients $e_{u,w}^{(k,t)} \in \mathbb{R}$ of neighbouring nodes defined by

$$e_{u,w}^{(k,t)} = \sigma(\mathbf{a}^{(k,t)^T} [\mathbf{W}^{(t)} \mathbf{h}_u \| \mathbf{W}^{(t)} \mathbf{h}_w]) \tag{2.10}$$

which in turn are each parameterised by a scoring vector $\mathbf{a}^{(k,t)} \in \mathbb{R}^{2d_2}$. Intuitively, GAT is able to perform better in learning tasks with more heterogenous graphs in which node neighbours are diverse in their features and labels due to the multi-head attention mechanism for selecting which features of neighbouring nodes are useful for updating embeddings, as opposed to the smoothing done by normalisation in GCN. However, they take significantly more time to train.

The final MPNN we cover is the Graph Isomorphism Network (GIN) [Xu et al., 2019] whose update equation is defined by

$$\mathbf{h}_u^{(t+1)} = \mathrm{MLP}^{(t)}\Big( (1 + \varepsilon^{(t)}) \mathbf{h}_u^{(t)} + \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(t)} \Big) \tag{2.11}$$

where again $\mathbf{h}_u^{(t)} \in \mathbb{R}^{d_2}, \mathbf{h}_u^{(t+1)} \in \mathbb{R}^{d_1}$, $\varepsilon^{(t)} \in \mathbb{R}$ is a learnable parameter, and $\text{MLP}^{(t)} : \mathbb{R}^{d_2} \mapsto \mathbb{R}^{d_1}$ denotes a multi-layer network generally implemented with one hidden layer. GIN belongs to the class of GNNs known as *spatial* GNNs which are motivated by local graph algorithms and convolutions, alongside GAT and GraphSage [Hamilton et al., 2017]. Spatial GNNs perform better on inductive tasks such as graph classification where input data have varying graph structure and size. The motivation for GIN is based on the Weisfeiler-Lehman algorithm for the graph isomorphism problem as we shall now discuss.

### 2.3.2 MPNNs and the Weisfeiler-Lehman algorithm

The graph isomorphism problem asks whether two given graphs are isomorphic. In other words, given $G = (V, E), H = (V', E')$, does there exist a bijection $\varphi : V \to V'$ such that $(u, v) \in E$ if and only if $(\varphi(u), \varphi(v)) \in E'$? The problem is in $\mathsf{NP}$ but it is not known whether it is $\mathsf{NP}$-complete or whether there is a polynomial time algorithm for solving it. Thus, it usually belongs to its own complexity class $\mathsf{GI}$.

The colour refinement algorithm or WL algorithm is an approximation for solving the problem by computing invariants for the graph. We refer the reader to [Cai et al., 1992] for a survey of approximation bounds. The algorithm is described in Alg. 6 and takes in a graph and returns a graph invariant with a multiset of integers representing colours. The main idea of the algorithm is to iteratively update the colour of each node by hashing the colours of the node itself and the multiset of nodes in its neighbourhood until the colours stabilise. This occurs in at most $|V| - 1$ iterations and the bound is tight [Kiefer and McKay, 2020].

**Algorithm 6:** Colour Refinement

**Data:** Graph $G = (V, E)$ and an injective hashing function $\varphi$ which maps a tuple of an integer and multiset of integers to an integer.

**Result:** Multiset of integers.

$c^{(0)}(v) \leftarrow 0, \forall v \in V$

**for** $i = 1, \ldots$ **do**

  $c^{(i)}(v) \leftarrow \varphi\left(c^{(i-1)}(v), \left\{\!\!\left\{c^{(i-1)}(u) \mid u \in \mathcal{N}(v)\right\}\!\!\right\}\right), \quad \forall v \in V$

  **if** $c^{(i)} = c^{(i-1)}$ **then**

    **return** $\left\{\!\!\left\{c^{(i)}(v) \mid v \in V\right\}\!\!\right\}$

An example of a set of graphs which the colour refinement algorithm cannot distinguish are $k$-regular graphs whose nodes share the same degree $k$ where the algorithm converges in one step. Fig. 2.2 illustrates the canonical pair of graphs which the WL algorithm cannot distinguish given that the algorithm sees the neighbourhood of each node in one graph as isomorphic to a node in the other.

So how does the WL algorithm relate to MPNNs and why does it matter? Firstly, one

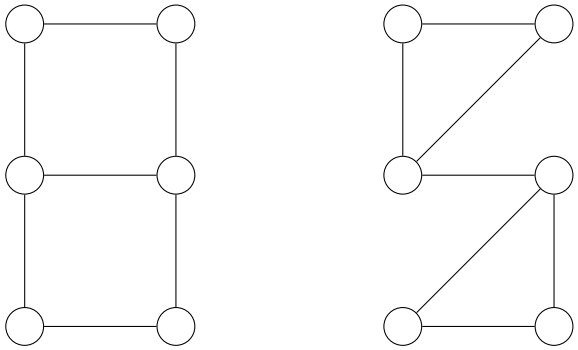


Figure 2.2: A pair of graphs which WL assigns the same output to.

can notice that the core ideas of the WL algorithm and MPNN are the same: both algorithms perform iterative updates on the information of each node on the graph based on its neighbours. Lem. 1 provides us a formal statement which tells us that the *expressiveness* of an MPNN is upper bounded by the WL algorithm. This in turn gives us a tool to measure what graphs that MPNNs cannot learn.

**Lemma 1** (MPNNs are as most as powerful as colour refinement [Xu et al., 2019])**.** *Let $G_1$ and $G_2$ be any two non-isomorphic graphs. If an MPNN maps $G_1$ and $G_2$ to two different embeddings, colour refinement assigns different invariants for $G_1$ and $G_2$.*

Colour refinement is known as a unique case of the general class of Weisfeiler-Lehman algorithms or $k$-WL algorithms [Cai et al., 1992] which iteratively update the colours of $k$-tuples of nodes $\mathbf{i} = (i_1, \dots, i_k) \in V^k$. Thus, colour refinement is otherwise known as the 1-WL algorithm. In the learning community, the $k$-WL algorithm usually known as the $k$-FWL algorithm [Morris et al., 2019, 2020] and the $k$-WL algorithm refers to a different procedure. Neighbourhoods for the $k$-FWL and $k$-WL algorithm are defined by

$$\mathcal{N}_j^{\mathrm{WL}}(\mathbf{i}) = \left\{ (i_1, \dots, i_{j-1}, u, i_{j+1}, \dots, i_k) \mid u \in [n] \right\} \tag{2.12}$$

$$\mathcal{N}_u^{\mathrm{FWL}}(\mathbf{i}) = \Big( (u, i_2, \dots, i_k), (i_1, u, \dots, i_k), \dots, (i_1, \dots, i_k, u) \Big) \tag{2.13}$$

and corresponding colour refinement updates are given by

$$\text{WL}: \quad c^{(l)}(\mathbf{i}) \leftarrow \varphi\left( c^{(l-1)}(\mathbf{i}), \left( \{\!\{ \zeta^{l-1}(\mathbf{j}) \mid \mathbf{j} \in \mathcal{N}_j^{\mathrm{WL}}(\mathbf{i}) \}\!\} \mid j \in [k] \right) \right) \tag{2.14}$$

$$\text{FWL}: \quad c^{(l)}(\mathbf{i}) \leftarrow \varphi\left( c^{(l-1)}(\mathbf{i}), \left\{\!\!\left\{ (\zeta^{l-1}(\mathbf{j}) \mid \mathbf{j} \in \mathcal{N}_u^{\mathrm{FWL}}(\mathbf{i})) \mid u \in [n] \right\}\!\!\right\} \right). \tag{2.15}$$

We also know that 1-WL distinguishes the same graphs as 2-WL, and that for $k > 1$, $k$-FWL distinguishes the same graphs as $k-1$-WL. Furthermore, the $k$-FWL algorithm is able to count substructures representable by $k$ variable first order logics [Cai et al.,

1992]. The result has been extended to GNNs [Barceló et al., 2020] and has been used to prove that it is possible to learn optimal policies for certain tractable planning domains which can be described using 2-variable counting logic [Staahlberg et al., 2022b].

### 2.3.3 Beyond MPNNs

One factor limiting the expressive power of MPNNs as we saw in Lem. 1 is their local update equation which means that a node's features rely only on the nodes in its $k$-hop neighbourhood in a $k$-layer MPNN which in turn prevents models from learning overall topological features of input graphs.

We are interested in expressive GNNs given that we are focused on applying them to learn methods for solving hard problems, such as planning whose simplest form in the propositional representation is PSPACE-complete [Bylander, 1994]. Specifically as we will see later, some graph representations and properties of planning problems are indistinguishable by simple MPNNs, i.e. an MPNN may output the same embedding for two semantically different planning problems.

Thus for the remainder of this subsection, we briefly discuss directions for building expressive GNNs whose models are founded on both theoretical and empirical results for graph representation learning.

#### Higher order GNNs

As discussed previously, there exists a generalisation of the colour refinement algorithm known as the $k$-WL algorithm which performs colour refinement on $k$-tuples of nodes as opposed to single nodes. There have been several works focused on building GNNs motivated by the $k$-WL algorithm [Morris et al., 2019, Maron et al., 2019, Morris et al., 2020, 2021a,b] but these generally come at a large computational and memory cost, exponential in $k$ (i.e. $O(n^k)$), and may not offer significant or even negative performance benefits on specific datasets [Dwivedi et al., 2020].

#### Injecting additional features

We may inject some additional precomputed information from the structure of the graph that a GNN cannot implicitly learn to aid in prediction. For example, it is known that cycles in molecular graphs are correlated to the outputs such as in the ZINC dataset where cycle counts are correlated with the target constrained solubility attribute of molecules [Irwin et al., 2012]. This motivates works which inject graph substructures [Bouritsas et al., 2022] and patterns [Barceló et al., 2021] into nodes. Overlap subgraph information can also be injected as edge features [Wijesinghe and Wang, 2022].

A complementary method is to inject random features [Sato et al., 2021, Abboud et al., 2021], with the intuition [Sato, 2020] being that nodes are able to distinguish other nodes in the receptive fields of the message passing scheme of a GNN and thus are able to distinguish substructures such as cycles that MPNNs cannot. Although this is a

simple method to implement with no computational overhead, generalisation to unseen graphs is poor.

Another method is to add distance encoding or positional encodings to nodes, similarly to what is done for Transformers [Vaswani et al., 2017]. One motivation for this is that graph neural networks may see the receptive field of two nodes in a graph as isomorphic but the two nodes may have different contributions to the overall graph. The method of considering distance or positional encodings of nodes has been done in the context of extending transformers to graphs [Dwivedi and Bresson, 2020] and encoding distances of nodes to a given set of target nodes in a graph [Li et al., 2020]. There have also been methods for learning distance information [You et al., 2019, Dwivedi et al., 2022]. Another natural method which generalises positional encoding of Transformers is by using graph Laplacian eigenvectors [Belkin and Niyogi, 2003] which has been shown to improve performance on various benchmarks [Dwivedi et al., 2020].

### Improving neighbourhood expressivity

A weakness of MPNNs is their simple local updates in which structure of node neighbourhoods are lost when they are aggregated and compressed with a permutation invariant function. The individualisation of nodes as done in ID-GNNs [You et al., 2021] is motivated by the fact that the computation tree of an MPNN cannot detect when the same node appears again and hence prevents us from detecting cycles. However, the method incurs an overhead by running a MPNN on each node of the graph such that the authors provide a fast version with the same complexity as MPNNs by injecting additional node information. This idea is not new in the GNN literature and has been used before to construct fast graph isomorphism solvers [McKay and Piperno, 2014]. GNN-AK [Zhao et al., 2022] and Nested GNN [Zhang and Li, 2021] propose to run an additional MPNN on the neighbourhood of each node instead of a shallow update and aggregate function. Again, this results in additional computation overhead so the authors of GNN-AK propose dropout on subgraphs. G3N [Wang et al., 2023] takes an orthogonal approach to $k$-WL algorithms by aggregating higher order objects within each node's neighbourhood to better compute local structural information without the prohibitive computation overhead of $k$-WL algorithms.

Chapter 3

# Graph representations

In this chapter, we begin our journey of constructing our GOOSE architecture by developing various novel graph representations of planning tasks. The graphs are constructed with domain-independent heuristic function learning in mind, with the motivation that we want our learning models to generalise over diverse tasks. In order to do so, we require encoding the full structure of planning tasks into our graphs.

This is in contrast to what is done in various other works in the learning for planning literature. More specifically, virtually all works which use graph representation learning focus on domain-dependent learning and do not encode the action schema into the structure of the graphs. They either implicitly learn the transition structure of planning tasks through the optimisation of a loss function or construct different set of weights and parameters corresponding to different various action schema. Tab. 3.1 summarises the graph representations for domain-independent learning that we study.

Table 3.1: Various graph representations of planning problems. Deletes indicate whether the graph representation encodes delete effects or not.

| Graph | New in this work | Lifted | Deletes | Edge labels | Undirected |
|---|---|---|---|---|---|
| DRG | ✓ | ✗ | ✗ | ✗ | ✗ |
| DRG$^{E}$ | ✓ | ✗ | ✗ | ✓ | ✓ |
| SDG | ✗ [Shleyfman et al., 2015] | ✗ | ✓ | ✗ | ✗ |
| SDG$^{E}$ | ✓ | ✗ | ✓ | ✓ | ✓ |
| FDG | ✗ [Pochter et al., 2011] | ✗ | ✓ | ✗ | ✓ |
| FDG$^{E}$ | ✓ | ✗ | ✓ | ✓ | ✓ |
| ASG | ✗ [Sievers et al., 2019] | ✓ | ✓ | ✗ | ✗ |
| LDG | ✓ | ✓ | ✓ | ✗ | ✓ |
| LDG$^{E}$ | ✓ | ✓ | ✓ | ✓ | ✓ |

We will use the definition and notation of a graph defined in Sec. 2.3 and further define a hypergraph in order to theoretically compare our work to STRIPS-HGN [Shen et al., 2020]. A hypergraph is a tuple $\langle V, E, \mathbf{X}, \mathbf{E} \rangle$ where again $V$ is a set of nodes, $E$ is a set of hyperedges $(A, B)$ where $A, B \subseteq V$ are non-empty subsets of nodes, $X : V \to \mathbb{R}^d$ are the node features and $E : E \to \mathbb{R}^e$ are the hyperedge features.

For simplicity, we assume the planning tasks we work with have unit action costs. This can also be done via compilation schemes which may introduce many additional actions and predicates relative to the action cost. However, we may also extend our framework to deal with general action costs by appending the action cost to features of action nodes. Encoding conditional action costs is not as straightforward or obvious.

## 3.1 Grounded graphs

In this section, we will look at graph representations of grounded STRIPS problems. This will be done in two steps. Firstly we will construct a representation analogous to the delete relaxation hypergraph constructed for STRIPS-HGN [Shen et al., 2020] and show that there exists a MPNN which can theoretically match the performance of STRIPS-HGN. The main motivation for doing this is so that we can shift our attention to graphs rather than hypergraphs, given that graphs are notationally and intuitively simpler to work with and do not have any clear disadvantage compared to hypergraphs for our setting. Following this we then move on to construct more expressive graphs which subsume the hypergraph definition from STRIPS-HGN.

### 3.1.1 STRIPS-HGN hypergraphs as graphs

To begin, let us define a natural graph representation of delete relaxed grounded STRIPS problems. The idea is that directions of edges represent information flow of action application and proposition activation. Features encode whether nodes represent either propositions or actions, and also whether a proposition is activated in the input state and/or is a goal condition.

**Definition 7** (Delete relaxation graph)**.** The *delete relaxation graph (DRG)* of a propositional STRIPS problem $\Pi = \langle P, A, s_0, G \rangle$ is the graph $\mathsf{DRG}(\Pi) = \langle V, E, \mathbf{X} \rangle$ with

- nodes $V = P \cup A$
- edges $E = \{(p, a) \in P \times A \mid p \in \text{pre}(a)\} \cup \{(a, p) \in A \times P \mid p \in \text{add}(a)\}$, and
- feature map $\mathbf{X} : V \to \mathbb{R}^4$ defined by

$$\begin{aligned} p &\mapsto \text{OH}(\textit{proposition}) + \text{OH}(\textit{activated?})(p) + \text{OH}(\textit{goal?})(p) && \text{if } p \in P \\ a &\mapsto \text{OH}(\textit{action}). && \text{if } a \in A \end{aligned}$$

  We define $\text{OH}(\textit{desc})$ as a one hot encoding of an enumeration of the description *desc*. For example for this definition we have the set of descriptions {*proposition*,

*activated*, *goal*, *action*} which we enumerate such that *proposition* is assigned 0, *activated* is assigned 1 and so on. Then OH(*proposition*) is a vector of size equal to the number of descriptions and is given by $[1, 0, 0, 0]$ with the 0-th coordinate assigned to 1.

We further define conditional one hot encoding functions as follows:

$$\text{OH}(\textit{activated?})(p) = \begin{cases} \text{OH}(\textit{activated}), & \text{if } p \in s_0, \\ \vec{0}, & \text{otherwise,} \end{cases}$$

$$\text{OH}(\textit{goal})(p) = \begin{cases} \text{OH}(\textit{goal}), & \text{if } p \in G, \\ \vec{0}, & \text{otherwise.} \end{cases}$$

Note that $(u, v) \in E \iff (v, u) \notin E$ as we assume $\text{pre}(a) \cap \text{add}(a) = \emptyset$ which we can enforce with a preprocessing step. ■

We can view the DRG as the unrolled planning graph of the Graphplan algorithm for the delete relaxation of a problem where facts point to actions they are a precondition of, and actions point to its add effects. However, one drawback of the above representation is that we have directed edges which limit the flow of information when used with an MPNN. The next chapter in Ch. 4 provides an illustrative example of the importance of information flow.

If we allow ourselves to define edge labels and use MPNNs which account for edge features or labels, we can construct a slightly more expressive graph for the task of inference. The idea is that we no longer require edge directions to encode whether a proposition is a precondition or add effect of an action as this can be done with edge labels instead.

**Definition 8** (Edge labelled delete relaxation graph)**.** The *edge labelled delete relaxation graph (DRG$^{\mathsf{E}}$)* of a propositional STRIPS problem $\Pi = \langle P, A, s_0, G \rangle$ is the undirected graph $\text{DRG}^{\mathsf{E}}(\Pi) = \langle V', E', \mathbf{X}', \mathbf{E}' \rangle$ where

- $V' = V$,
- $E' = \{\{u, v\} \mid (u, v) \in E\}$,
- $\mathbf{X}' = \mathbf{X}$,
- $\mathbf{E}' : E' \to \{\text{pre}, \text{add}\}$ where $\mathbf{E}'(e) = \text{pre}$ if $e = \{p, a\}$ with $p \in \text{pre}(a)$ otherwise $\mathbf{E}'(e) = \text{add}$,

and $V$, $E$, and $\mathbf{X}$ are the objects defined in the delete relaxation graph $\text{DRG}(\Pi) = \langle V, E, \mathbf{X} \rangle$. ■

To understand how our definitions so far are related to the STRIPS-HGN framework, we define the underlying hypergraph structure used for STRIPS-HGN. We refer to their implementation for the features[1]. However, Shen et al. [2020] mentioned in their paper

[1]Found at https://github.com/williamshen-nz/STRIPS-HGN.

that additional precomputed features can be appended such as encoding whether a proposition or action node is a landmark which may also be done similarly in our context.

**Definition 9** (Delete relaxation hypergraph [Shen et al., 2020])**.** The *delete relaxation hypergraph (DRH)* of a propositional STRIPS problem $\Pi = \langle P, A, s_0, G\rangle$ is the hypergraph $\mathsf{DRH}(\Pi) = \langle V, E, \mathbf{X}, \mathbf{E}\rangle$ with

- $V = P$,
- $E = \{(\text{pre}(a), \text{add}(a)) \mid a \in A\}$,
- $\mathbf{X} : V \to \mathbb{R}^2$ defined by $\mathbf{X}(p) = [i, j]$ where $i = 1$ if $p \in s_0$ else 0, and $j = 1$ if $j \in G$ else 0,
- $\mathbf{E} : E \to \mathbb{R}^2$ defined by[2] $\mathbf{X}(a) = [|\text{pre}(a)|, |\text{add}(a)|]$. ■

Next, we briefly describe the update function of the hypergraph network architecture. In each iteration, we keep track of 3 sets of vectors: a global feature vector $\boldsymbol{u}$ which is analogous to a virtual node, a set of node feature vectors $\mathbf{v}_i$ for each node $i \in V$, and a set of hyperedge feature vectors $\mathbf{e}_k$ for each hyperedge $k \in E$. Then these features are iteratively updated $L$ times with the *same* HGN-block (message passing layer) with update functions $\varphi^e$, $\varphi^v$ and $\varphi^u$ by

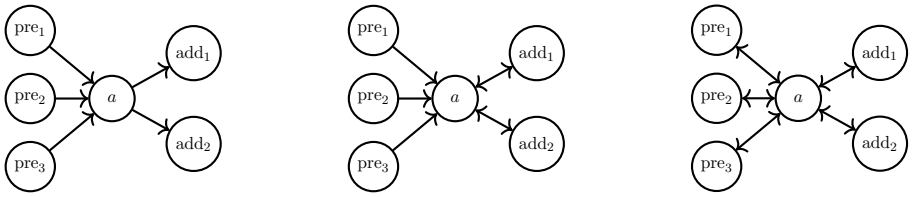
$$\begin{aligned}
\mathbf{e}_k^{(t+1)} &= \varphi^e(\mathbf{e}_k^{(t)}, \mathbf{R}_k^{(t)}, \mathbf{S}_k^{(t)}, \boldsymbol{u}^{(t)}) \\
\mathbf{v}_i^{(t+1)} &= \varphi^e(\overline{\mathbf{e}}_i^{(t+1)}, \mathbf{v}_i^{(t)}, \boldsymbol{u}^{(t)}) \\
\mathbf{u}^{(t+1)} &= \varphi^e(\overline{\mathbf{e}}^{(t+1)}, \overline{\mathbf{v}}^{(t+1)}, \boldsymbol{u}^{(t)})
\end{aligned} \tag{3.1}$$

where $\mathbf{R}_k = \{\mathbf{v}_j \mid j \in R_k\}$ denotes the vectors of the *receivers*[3] $R_k$ of hyperedge $k$ and $\mathbf{S}_k = \{\mathbf{v}_j \mid j \in S_k\}$ the *senders*[4] $S_k$. The vectors with bars over them denote aggregated vectors given by

$$\begin{aligned}
\overline{\mathbf{e}}_i^{(t+1)} &= \square_{e\to v}(\{\mathbf{e}_k^{(t)} \mid i \in R_k\}) \\
\overline{\mathbf{e}}^{(t+1)} &= \square_{e\to u}(\{\mathbf{e}_k \mid k \in E\}) \\
\overline{\mathbf{v}}^{(t+1)} &= \square_{v\to u}(\{\mathbf{v}_i \mid i \in V\})
\end{aligned} \tag{3.2}$$

where $\square_\alpha$ are permutation aggregation functions with subscripts $\alpha$ marking possibly different functions. An important point to recognise in the definition of the STRIPS-HGN block is that the update function $\varphi^e$ takes in two sets $\mathbf{R}_k$ and $\mathbf{S}_k$. However, $\varphi^e$ is defined by ordering the vectors in $\mathbf{R}_k$ and $\mathbf{S}_k$ by their corresponding proposition names and concatenating them alongside $\mathbf{e}_k$ and $\mathbf{u}$ before feeding them into an MLP. To deal with variable sized sets, STRIPS-HGN assumes an upper bound on the size of $\mathbf{R}_k$ and $\mathbf{S}_k$ and deals with smaller sets by padding with zeros. The main takeaways from this construction are that STRIPS-HGN:

[2] The cost of actions is also encoded but this is excluded due to our assumption of unit action costs, which also the case for STRIPS-HGN.
[3] Add effects of the action corresponding to the hyperedge.
[4] Preconditions of the action corresponding to the hyperedge.




(a) DRG & MPNN (b) STRIPS-HGN (c) DRG$^{\mathsf{E}}$ & MPNN

Figure 3.1: Information flow of delete relaxation representations DRG and DRG$^{\mathsf{E}}$ with MPNNs and STRIPS-HGNs represented with arrows. Information flow from global graph features or virtual nodes are omitted.

- is ***not*** permutation invariant to renaming of propositions, which may increase domain dependent heuristic performance but limits its generalisation power over multiple domains, and
- assumes a maximum size of action preconditions and effects, meaning that it may discard information of certain domains when performing domain-independent training and evaluation.

Fig. 3.1 illustrates the direction of information flow of STRIPS-HGN (with global features omitted) compared to MPNNs acting on DRG and DRG$^{\mathsf{E}}$, where we note that Eq. 3.1 provides additional information flow in STRIPS-HGN over DRG while Eq. 3.2 is the cause for restricted information flow in comparison to DRG$^{\mathsf{E}}$.

With the definition of the core components of the STRIPS-HGN architecture complete, we can formalise how it is possible to compile STRIPS-HGN into a simpler framework acting on graphs and not hypergraphs. Given a set of parameters $\boldsymbol{\Theta}_H$, we denote $\mathcal{F}^{\mathsf{STRIPS\text{-}HGN}}_{\boldsymbol{\Theta}_H}$ a STRIPS-HGN model initialised with these parameters which takes in as input a planning task and outputs a heuristic value.

**Proposition 1.** *Given a STRIPS-HGN instantiation $\mathcal{F}^{\mathsf{STRIPS\text{-}HGN}}_{\boldsymbol{\Theta}_H}$, there exists a set of parameters $\boldsymbol{\Theta}_G$ for a non permutation-invariant MPNN with a virtual node $\mathcal{F}_{\boldsymbol{\Theta}_G}$ such that for all planning tasks $\Pi = \langle P, A, s_0, G\rangle$ we have $\mathcal{F}_{\boldsymbol{\Theta}_G}(\mathsf{DRG}^{\mathsf{E}}(\Pi)) = \mathcal{F}^{\mathsf{STRIPS\text{-}HGN}}_{\boldsymbol{\Theta}_H}(\Pi)$.*

*Proof sketch.* The main idea of the proof is that the information flow in STRIPS-HGN is weaker than that of an MPNN operating on DRG$^{\mathsf{E}}$ as illustrated in Fig. 3.1. The virtual node is required for an MPNN to emulate the global feature $\boldsymbol{u}$ in STRIPS-HGN. We note that the MPNN will require double the number of layers of STRIPS-HGN to mimic the hypergraph message passing architecture in this way. We can also encode an aggregation function for the MPNN which mimics the one in STRIPS-HGN, noting that it is not permutation-invariant due to the sorting and concatenation of neighbouring nodes. The aggregation function can also be crafted to encode node degrees to mimic the encoded node features of STRIPS-HGN. Thus, we have that an MPNN acting on DRG$^{\mathsf{E}}$ can mimic any initialisation of STRIPS-HGN due to its stronger information flow and equivalent feature information. □

### 3.1.2 Grounded STRIPS graphs with full information

One caveat of the STRIPS-HGN architecture is that it doesn't encode the full information of the input planning problems. Namely, it ignores delete lists. This is restrictive in both its expressivity and generalisation capabilities given that it discards useful information of the structure of the problems. Furthermore, since it is trained on optimal heuristic values, it may overfit on unnecessary structure of the hypergraphs to account for the missing information. Thus in this section we look at graphs which do not discard information of our planning problems, some which we modify from the literature and others which we construct to better fit our learning task.

A graph representation for grounded STRIPS problems already exists, namely the STRIPS problem description graph [Shleyfman et al., 2015]. It was originally used to study which classical heuristics were invariant under symmetries in the planning task. We modify the definition of the STRIPS problem description graph with node features and additional edges marked bold below to better fit our learning framework and theory but name it in the same way. We also note that there is a different problem description graph for FDR problems [Pochter et al., 2011] which we will explore shortly.

**Definition 10** (STRIPS problem description graph [Shleyfman et al., 2015])**.** The *problem description graph (SDG)* of a propositional STRIPS problem $\Pi = \langle P, A, s_0, G\rangle$ is the graph $G = \langle V, E, \mathbf{X}\rangle$ with

- $V = A \cup \bigcup_{p\in P}\{p, p^T, p^F\}$
- $E = \bigcup_{p\in P}\{(p, p^T), (p, p^F), (\mathbf{p^F}, \mathbf{p^T})\} \cup \bigcup_{a\in A}(E_a^{\text{pre}} \cup E_a^{\text{add}} \cup E_a^{\text{del}})$ with
  - $E_a^{\text{pre}} = \big\{(p^T, a) \mid p \in \text{pre}(a)\big\}$,
  - $E_a^{\text{add}} = \big\{(a, p^T) \mid p \in \text{add}(a)\big\}$,
  - $E_a^{\text{del}} = \big\{(a, p^F) \mid p \in \text{del}(a)\big\}$
- $\mathbf{X} : V \to \mathbb{R}^5$ defined by

$$\begin{aligned} p &\mapsto \text{OH}(\textit{activated?})(p) + \text{OH}(\textit{goal?})(p) && \text{if } p \in P \\ p^T &\mapsto \text{OH}(T) && \text{if } p \in P \\ p^F &\mapsto \text{OH}(F) && \text{if } p \in P \\ a &\mapsto \text{OH}(\textit{action}) && \text{if } a \in A. \end{aligned}$$

■

We note that SDG encodes information about delete lists with auxiliary proposition nodes and additional edges connecting how the actions interact with the propositions. Without the additional edges marked in bold, MPNNs would be blind to such delete effects. It is possible to make a description graph which uses auxiliary action nodes instead of auxiliary proposition nodes, representing the add and delete effects, but note

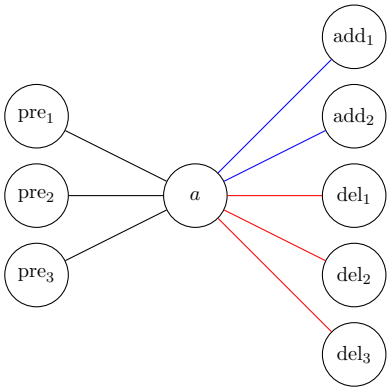



Figure 3.2: $\mathsf{SDG}^{\mathsf{E}}$ subgraph of an action $a$ with $\text{pre}(a) = \{\text{pre}_1, \text{pre}_2, \text{pre}_3\}$, $\text{add}(a) = \{\text{add}_1, \text{add}_2\}$ and $\text{del}(a) = \{\text{del}_1, \text{del}_2, \text{del}_3\}$. In the case where a proposition is a precondition and also in the delete effect, we will have a multiedge between the proposition and action node.

that this may not be preferred given that in practice the number of actions in a planning problem is greater than the number of propositions.

A limiting problem of this encoding is that it takes several message passing steps for an MPNN to understand the semantics of adding or deleting a proposition and thus limits their expressivity as MPNNs have a fixed number of layers and receptive field. Furthermore, the directed edges further limit the information flow and expressivity when used with MPNNs. If we allow for edge features, we may remove the need for such auxiliary nodes, reduce the size of the graph and allow for undirected graphs. Fig. 3.2 illustrates the following definition with the subgraph induced by a single action.

**Definition 11** (Edge-labelled STRIPS problem description graph)**.** The *edge-labelled problem description graph (*$\mathsf{SDG}^{\mathsf{E}}$*)* of a propositional STRIPS problem $\Pi = \langle P, A, s_0, G\rangle$ is the undirected multigraph $G = \langle V, E, \mathbf{X}, \mathbf{E}\rangle$ with

- $V = A \cup P$
- $E = E^{\text{pre}} \cup E^{\text{add}} \cup E^{\text{del}}$ with
  - $E^{\text{pre}} = \{(\{p, a\}, \text{pre}) \mid p \in \text{pre}(a), a \in A\}$,
  - $E^{\text{add}} = \{(\{a, p\}, \text{add}) \mid p \in \text{add}(a), a \in A\}$,
  - $E^{\text{del}} = \{(\{a, p\}, \text{del}) \mid p \in \text{del}(a), a \in A\}$
- $\mathbf{X} : V \to \mathbb{R}^4$ defined by

$$\begin{aligned} p &\mapsto \text{OH}(\textit{proposition}) + \text{OH}(\textit{activated?})(p) + \text{OH}(\textit{goal?})(p) && \text{if } p \in P \\ a &\mapsto \text{OH}(\textit{action}) && \text{if } a \in A. \end{aligned}$$

- $\mathbf{E} : E \to \{\text{pre}, \text{add}, \text{del}\}$ defined by $e = (\{u, v\}, i) \mapsto i$ for $i = \text{pre}, \text{add}, \text{del}$. ■

### 3.1.3 FDR graphs

As we have discussed in Sec. 2.1, planners usually translate planning tasks to FDR problems which compute mutexes in the problem. We start with the problem description graph for FDR problems [Pochter et al., 2011] which was originally utilised for computing symmetries in order to speed up search by pruning symmetrical states. We modify the original definition to better fit our learning task again by introducing node features by taking one hot encodings of the node types and encoding the current state and goal condition. We assume for ease of notation that the $D_v$ for any FDR problem are pairwise disjoint which can be enforced by prefixing values in $D_v$ with the variable $v$.

**Definition 12** (FDR problem description graph [Pochter et al., 2011])**.** The *problem description graph (FDG)* of an FDR problem $\Pi = \langle \mathcal{V}, A, s_0, s_\star \rangle$ is the undirected graph $G = \langle V, E, \mathbf{X} \rangle$ with

- $V = \mathcal{V} \cup \bigcup_{v \in \mathcal{V}} D_v \cup \bigcup_{a \in A} \{a_{\text{pre}}, a_{\text{eff}}\}$
- $E = E_{\text{var:val}} \cup E_{\text{pre}} \cup E_{\text{eff}} \cup E_{\text{act}}$ where
  - $E_{\text{var:val}} = \bigcup_{v \in \mathcal{V}} \{\{v, d\} \mid d \in D_v\}$
  - $E_{\text{pre}} = \bigcup_{a \in A} \{\{d, a_{\text{pre}}\} \mid (v, d) \in \text{pre}(a)\}$
  - $E_{\text{eff}} = \bigcup_{a \in A} \{\{d, a_{\text{eff}}\} \mid (v, d) \in \text{eff}(a)\}$
  - $E_{\text{act}} = \{\{a_{\text{pre}}, a_{\text{eff}}\}\}$
- $\mathbf{X} : V \to \mathbb{R}^6$ defined by

$$
\begin{aligned}
v &\mapsto \text{OH}(\textit{var}) && \text{if } v \in \mathcal{V} \\
d &\mapsto \text{OH}(\textit{val}) + \text{OH}(\textit{activated?})(v) + \text{OH}(\textit{goal?})(v) && \text{if } \exists v \in \mathcal{V}, d \in D_v \\
a_{\text{pre}} &\mapsto \text{OH}(\textit{pre}) && \text{if } a \in A \\
a_{\text{eff}} &\mapsto \text{OH}(\textit{eff}) && \text{if } a \in A.
\end{aligned}
$$

■

Similarly to what we have done for the grounded STRIPS graphs, we construct a variant of FDG with edge labels in order the better aid the GNN understand the semantics of the problem and to reduce the size of the graph. The main difference we introduce is that each action now has only one associated node, instead of two in FDG, to represent the difference between preconditions and effects, as this is encoded in the edge labels. We may assume that preconditions are disjoint from effects for FDR problems such that we do not require a multigraph as with $\text{SDG}^{\text{E}}$. Fig. 3.3 illustrates the edge labelled variant $\text{FDG}^{\text{E}}$.

**Definition 13** (Edge-labelled FDR problem description graph)**.** The *edge-labelled problem description graph ($\text{FDG}^{\text{E}}$)* of an FDR problem $\Pi = \langle \mathcal{V}, A, s_0, s_\star \rangle$ is the undirected graph $G = \langle V, E, \mathbf{X}, \mathbf{E} \rangle$ with

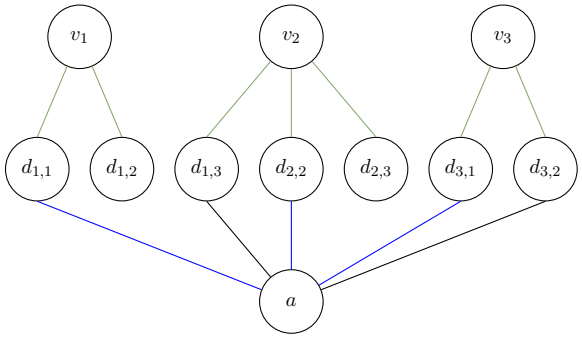



Figure 3.3: $\mathsf{FDG}^{\mathsf{E}}$ subgraph of an action $a$ with $\text{pre}(a) = \{\langle v_2, d_{2,1}\rangle, \langle v_3, d_{3,2}\rangle\}$ and $\text{eff}(a) = \{\langle v_1, d_{1,1}\rangle, \langle v_2, d_{2,2}\rangle, \langle v_3, d_{3,1}\rangle\}$.

- $V = \mathcal{V} \cup \bigcup_{v \in \mathcal{V}} D_v \cup A$
- $E = E_{\text{var:val}} \cup E_{\text{pre}} \cup E_{\text{eff}}$ where
  - $E_{\text{var:val}} = \bigcup_{v \in \mathcal{V}} \{\{v, d\} \mid d \in D_v\}$
  - $E_{\text{pre}} = \bigcup_{a \in A} \{\{d, a\} \mid (v, d) \in \text{pre}(a)\}$
  - $E_{\text{eff}} = \bigcup_{a \in A} \{\{d, a\} \mid (v, d) \in \text{eff}(a)\}$
- $\mathbf{X} : V \to \mathbb{R}^5$ defined by

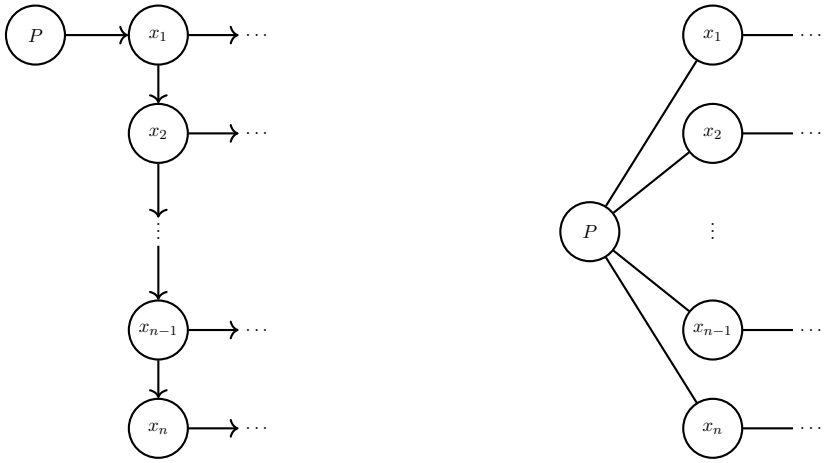
$$\begin{aligned} v &\mapsto \text{OH}(\textit{var}) && \text{if } v \in \mathcal{V} \\ d &\mapsto \text{OH}(\textit{val}) + \text{OH}(\textit{activated?})(v) + \text{OH}(\textit{goal?})(v) && \text{if } \exists v \in \mathcal{V}, d \in D_v \\ a &\mapsto \text{OH}(\textit{action}) && \text{if } a \in A \end{aligned}$$

- $\mathbf{E} : E \to \{\text{var:val}, \text{pre}, \text{eff}\}$ where $e \to i$ if $e \in E_i$ for $i \in \{\text{var:val}, \text{pre}, \text{eff}\}$. ■

## 3.2 Lifted graphs

As we have discussed previously, there is much motivation to construct algorithms that can perform planning based on the lifted representation without the need for grounding all predicates and action schema in order to save memory. Constructing a sensible graph representation for a lifted task for the objective of learning is not as easy as in the grounded case as there are more relations to encode, namely the interactions between predicates, action schema, propositions true in the current state, the goal condition and objects, in comparison to simply propositions and actions in grounded tasks.

The only graph representation encoding all the information of a planning task in its lifted form is the *abstract structure graph* (ASG) [Sievers et al., 2019] which is defined by first defining a coloured graph on *abstract structures*, a recursive structure defined with sets, tuples, and the input objects, and then defining a lifted planning task as an




(a) ASG encoding of predicate arguments (b) LDG encoding of predicate arguments

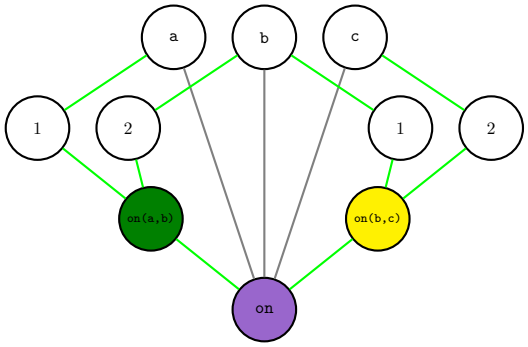
Figure 3.4: Encodings of predicate arguments with different lifted graph representations.

abstract structure. The original intent of the ASG was to compute symmetries, similarly to FDG, but it has also been used for learning planning portfolios [Katz et al., 2018].

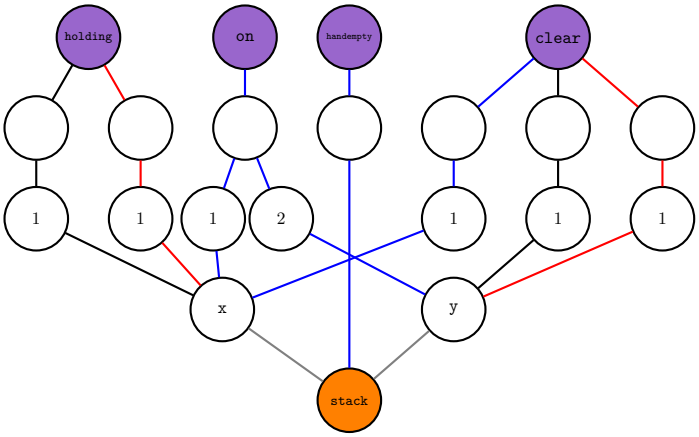
Now we discuss the main limitations of ASGs for MPNNs. Firstly, the encoding of predicate and action schema arguments is done via a sequence or directed path, where to encode $n$ arguments the graph consists of a directed path of length $n$, as illustrated in Fig. 3.4a. There are also many more auxiliary nodes to encode the abstract structures. Both these issues cause problems for MPNNs as we have seen before for the grounded graph representations in the literature: larger receptive field required for MPNNs to learn the structure and semantics of the planning problem, and directed edges which limit information flow and expressivity.

Thus, we attempt to construct our own graph representations for lifted planning tasks with the objective of learning in conjunction with GNNs. We first provide a verbose description and illustration of the edge-labelled variant of the lifted graphs in Fig. 3.5 before providing the full formal definition. The graph can be divided into two main subgraphs, the first of which encodes the current state and the goal condition of a lifted planning task, and the second encodes the underlying action schema of the task. In both subgraphs, we require encoding predicate or action schema arguments which we solve by using node features rather than using the structure of the graph to encode argument indexes, allowing us to represent predicates with trees rather than paths as illustrated in Fig. 3.4b.

The first subgraph contains nodes representing the predicates and objects of the problem. We introduce additional nodes of the form described in Fig. 3.4b for encoding the grounded propositions that are true in the current state and the goal condition. Various edges are used to associate the corresponding objects and predicates for each grounded proposition as seen in Fig. 3.5a.




(a) LDG$^{\mathsf{E}}$ subgraph of a ground `on(a,b)` predicate activated in the current state and an `on(b,c)` goal predicate in the Blocksworld domain.




(b) LDG$^{\mathsf{E}}$ subgraph of the `stack(x,y)` action schema in the Blocksworld domain.

Figure 3.5: LDG$^{\mathsf{E}}$ subgraph of ground predicates (a) and an action schema (b) with graph layer descriptions. The underlying graph structure of LDG$^{\mathsf{E}}$ is isomorphic to that of LDG.

The second subgraph contains the same aforementioned predicate nodes and a node for each action schema. We then use auxiliary nodes to act as duplicates of predicates seen in the precondition and effects of each action schema, given that action schema can have several uses of a predicate in its definition. We also make use of the above argument encodings and various edges to match action schema arguments with predicate arguments as seen in Fig. 3.5b for the `stack(x,y)` schema in the Blocksworld domain.

Now we move onto the formal definition. We will assume that action schema preconditions and effect lists only contain parameter variables and no objects for simplicity. In other words, we do not have partially instantiated action schema, and this is generally the case for most PDDL files. We also assume no negative preconditions and negated atoms in the goal state, but we implemented this feature in the code which requires an additional node feature dimension and edge label.

**Definition 14** (Lifted description graph)**.** Let $T \in \mathbb{N}$. The *lifted problem description graph (LDG)* of a lifted problem $\Pi = \langle \mathcal{P}, \mathcal{O}, \mathcal{A}, s_0, G \rangle$ is the undirected graph $G = \langle V, E, \mathbf{X} \rangle$ with[5]

- $V = \mathcal{P} \cup \mathcal{O} \cup N(\mathcal{A}) \cup N(s_0) \cup N(G)$ where
  - $N(\mathcal{A})$ provides all the nodes corresponding to the action schema, schema arguments, predicate arguments and schema predicates layers as depicted in Fig. 3.5b and is defined by

$$N(\mathcal{A}) = \bigcup_{a \in \mathcal{A}} \Bigg( \{a\} \cup \{(\delta, a) \mid \delta \in \Delta(a)\} \cup \bigcup_{f \in \{\text{pre,add,del}\}} \Big\{ (p, a, f), (1, p, a, f), \ldots, (n_P, p, a, f) \mid p = P(\delta_1, \ldots, \delta_{n_P}) \in f(a) \Big\} \Bigg) \tag{3.3}$$

  - $N(s_0)$ and $N(G)$ provide nodes corresponding to the state and goal, and ground arguments layer as in Fig. 3.4b and is defined by

$$N(s_0) = \bigcup_{p = P(o_1, \ldots, o_{n_P}) \in s_0} \{(p, 0), (1, p, 0), \ldots, (n_P, p, 0)\} \tag{3.4}$$

$$N(G) = \bigcup_{p = P(o_1, \ldots, o_{n_P}) \in G} \{(p, g), (1, p, g), \ldots, (n_P, p, g)\} \tag{3.5}$$

- $E = E_{\text{neutral}} \cup E_{\text{ground}} \cup \bigcup_{f \in \{\text{pre,add,del}\}} E_f$ where
  - $E_{\text{neutral}}$ connects objects to predicates and actions to schema arguments, indicated by gray edges in Fig. 3.5 and is defined by

$$E_{\text{neutral}} = \Big\{ \{o, P\} \mid o \in \mathcal{O}, P \in \mathcal{P} \Big\} \cup \Big\{ \{a, (\delta, a)\} \mid \delta \in \Delta(a), a \in \mathcal{A} \Big\} \tag{3.6}$$

  - $E_{\text{ground}}$ connects nodes in $\mathcal{P}$, $\mathcal{O}$, $N(s_0)$ and $N(G)$ in order to represent propositions in the goal and true in the state as instantiated predicates with objects in the correct arguments. This set of edges is defined by

[5] The definition is slightly different when handling negative preconditions where we have an additional node feature and edge label for the edge labelled version of the graph.

$$E_{\text{ground}} = \bigcup_{p=P(o_1,\ldots,o_{n_P})\in s_0} \Bigg( \Big\{ \{(p,0),(i,p,0)\} \mid i=1,\ldots,n_P \Big\} \cup \Big\{ \{(i,p,0),o_i\} \mid i=1,\ldots,n_P \Big\} \cup \{(p,0),P\} \Bigg) \cup \bigcup_{p=P(o_1,\ldots,o_{n_P})\in G} \Bigg( \Big\{ \{(p,g),(i,p,g)\} \mid i=1,\ldots,n_P \Big\} \cup \Big\{ \{(i,p,g),o_i\} \mid i=1,\ldots,n_P \Big\} \cup \{(p,g),P\} \Bigg) \tag{3.7}$$

– $\bigcup_{f\in\{\text{pre,add,del}\}} E_f$ provides the set of edges connecting nodes in $\mathcal{P}$ and $N(\mathcal{A})$ to encode the semantics of action schema in the graph. The individual $E_f$ components are defined by

$$E_f = \bigcup_{p=P()\in f(a)} \Big\{ \{P,(p,a,f)\}, \{(p,a,f),a\} \Big\} \cup \bigcup_{p=P(\delta_1,\ldots,\delta_{n_P})\in f(a), n_P\geq 1} \Bigg( \Big\{ \{P,(p,a,f)\} \Big\} \cup \Big\{ \{(p,a,f),(i,p,a,f)\}, \{(i,p,a,f),(\delta_i,a)\} \mid i=1,\ldots,n_P \Big\} \Bigg) \tag{3.8}$$

for $f \in \{\text{pre},\text{add},\text{del}\}$,

- $\mathbf{X} : V \to \mathbb{R}^{8+T}$ defined by

$$\begin{aligned}
P &\mapsto \mathrm{OH}(\textit{predicate}) \,\|\, \vec{0} && \text{for } P\in\mathcal{P}\\
o &\mapsto \mathrm{OH}(\textit{object}) \,\|\, \vec{0} && \text{for } o\in\mathcal{O}\\
a &\mapsto \mathrm{OH}(\textit{action}) \,\|\, \vec{0} && \text{for } a\in\mathcal{A}\\
(p,a,f) &\mapsto \mathrm{OH}(f) \,\|\, \vec{0} && \text{for } f\in\{\text{pre},\text{add},\text{del}\}, p=P(\delta_1,\ldots,\delta_{n_P})\in f(a)\\
(i,p,a,f) &\mapsto \vec{0} \,\|\, \mathrm{PE}(i) && i=1,\ldots,n_P\\
(p,g) &\mapsto \mathrm{OH}(\textit{goal}) \,\|\, \vec{0} && \text{for } p=P(o_1,\ldots,o_{n_p})\in G\\
(i,p,g) &\mapsto \vec{0} \,\|\, \mathrm{PE}(i) && \text{for } i=1,\ldots,n_P\\
(p,0) &\mapsto \mathrm{OH}(\textit{activated}) \,\|\, \vec{0} && \text{for } p=P(o_1,\ldots,o_{n_p})\in s_0\\
(i,p,0) &\mapsto \vec{0} \,\|\, \mathrm{PE}(i) && \text{for } i=1,\ldots,n_P
\end{aligned}$$

and $v \mapsto \vec{0}$ for any remaining nodes, i.e. nodes of the form $(\delta,a)$ for $a\in\mathcal{A}, \delta\in\Delta(a)$. We define $\mathrm{PE}:\mathbb{N}\to\mathbb{R}^T$ by a fixed randomly chosen injective map from $\mathbb{N}$ to the sphere $S^{T-1} = \{x\in\mathbb{R}^T \mid \|x\|=1\}$. ■

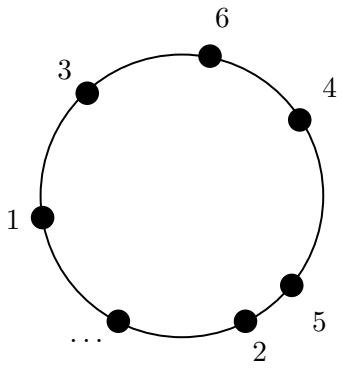



Figure 3.6: An example of a PE function for $T = 2$.

The function PE is constructed with domain-independence in mind, meaning that we cannot use information about domains to construct our graphs as this allows us to incorporate domain-independent or multi-domain learning. We can contrast this to Neural Logic Machines [Dong et al., 2019] which were designed for use on specified domains and thus can encode argument indices in a more structured way. The domain-independence motivations we used to guide the construction of PE are:

1. we cannot assume a bound on predicate and action schema argument sizes, and
2. the argument indices should be independent from one another

The first motivation rules out using a one hot encoding of indices as otherwise we will have infinite dimensional feature vectors. The second motivation makes it difficult to construct a well defined deterministic feature map such as the sinusoidal positional encodings used in Transformers [Vaswani et al., 2017]. Thus, we opted with nondeterministically constructing such an injective function PE which is fixed for *all* domains and instances. Besides satisfying the 2 motivations we described, it is unclear if there exists a better method of encoding argument indices via graphs. We now present the edge labelled variant of LDG.

**Definition 15** (Edge-labelled lifted description graph)**.** The *edge-labelled lifted problem description graph (LDG$^E$)* of a lifted problem $\Pi = \langle \mathcal{P}, \mathcal{O}, \mathcal{A}, s_0, G \rangle$ is the undirected graph $G = \langle V, E, \mathbf{X}, \mathbf{E} \rangle$ with $V$ and $E$ the same as in LDG and

- $\mathbf{X} : V \to \mathbb{R}^{5+T}$ with the same definition as in LDG except now we have $(p, a, f) \mapsto 0$ for $a \in \mathcal{A}, f \in \{\text{pre}, \text{add}, \text{del}\}, p \in f(a)$.
- $\mathbf{E} : E \to \{\text{neutral}, \text{ground}, \text{pre}, \text{add}, \text{del}\}$ defined by $e \mapsto \alpha$ for $e \in E_\alpha$ with $\alpha \in \{\text{neutral}, \text{ground}, \text{pre}, \text{add}, \text{del}\}$. ■

We conclude this chapter with complete illustrations in Fig. 3.7 of some of the defined graph representations for a Blocksworld problem with PPDL definition in Lst. 3.1 and 3.2. We also refer to the Appendix for boxplots of graph sizes of the individual graph representations in Sec. A.1.

Listing 3.1: Blocksworld PDDL domain.

```
(define (domain blocks)
  (:requirements :strips :typing)

  (:types block)

  (:predicates
    (on ?x - block ?y - block)
    (ontable ?x - block)
    (clear ?x - block)
    (handempty)
    (holding ?x - block)
  )

  (:action pick-up
    :parameters (?x - block)
    :precondition (and (clear ?x)
                       (ontable ?x)
                       (handempty))
    :effect (and (not (ontable ?x))
                 (not (clear ?x))
                 (not (handempty))
                 (holding ?x))
  )

  (:action put-down
    :parameters (?x - block)
    :precondition (holding ?x)
    :effect (and (not (holding ?x))
                 (clear ?x)
                 (handempty)
                 (ontable ?x))
  )

  (:action stack
    :parameters (?x - block ?y - block)
    :precondition (and (holding ?x)
                       (clear ?y))
    :effect (and (not (holding ?x))
                 (not (clear ?y))
                 (clear ?x)
                 (handempty)
                 (on ?x ?y))
  )

  (:action unstack
    :parameters (?x - block ?y - block)
    :precondition (and (on ?x ?y)
                       (clear ?x)
                       (handempty))
    :effect (and (holding ?x)
                 (clear ?y)
                 (not (clear ?x))
                 (not (handempty))
                 (not (on ?x ?y)))
  )
)
```

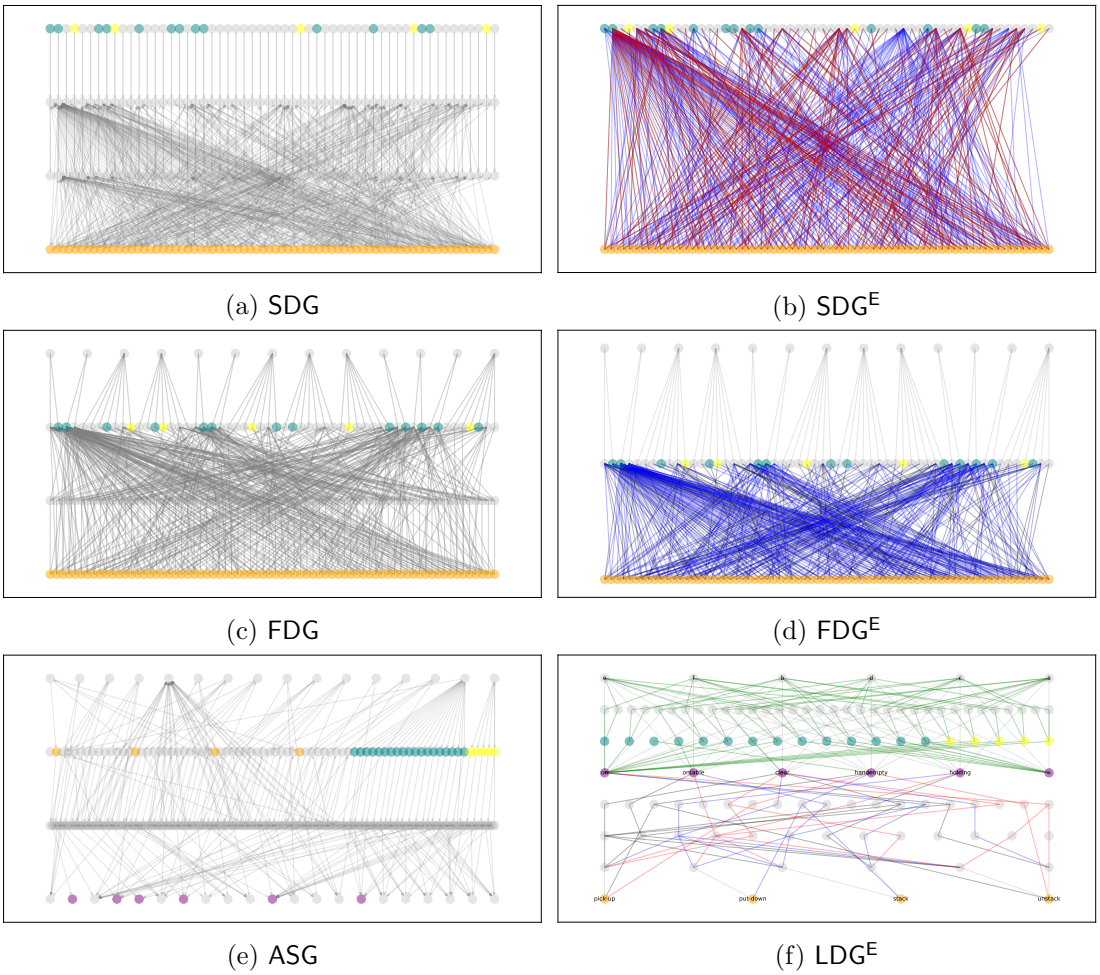
Listing 3.2: Blocksworld PDDL instance.

```
(define (problem blocks-6-2)
  (:domain blocks)

  (:objects a b c d e f - block)

  (:init
    (clear a)
    (ontable c)
    (on a d)
    (on d b)
    (on b f)
    (on f e)
    (on e c)
    (handempty)
  )

  (:goal (and (on e f)
              (on f a)
              (on a b)
              (on b c)
              (on c d))
  )
)
```



(a) SDG (b) SDG$^{\mathsf{E}}$

(c) FDG (d) FDG$^{\mathsf{E}}$

(e) ASG (f) LDG$^{\mathsf{E}}$

Figure 3.7: Graph representations of the Blocksworld instance described in Lst. 3.1 and 3.2. LDG is omitted as it is structurally the same as LDG$^{\mathsf{E}}$ but without edge labels. Graphs on the right have edge labels. Green nodes correspond to facts true in the current state. Yellow nodes correspond to goal facts. Orange nodes correspond to grounded or lifted actions. Purple nodes correspond to predicates. Black, blue and red edges correspond to preconditions, add effects and delete effects respectively.

Chapter 4

# What can we learn?

We have defined a whole zoo of graphs for representing planning problems in Ch. 3 with a focus on learning domain-independent heuristics when combined with graph representation learning methods. In this chapter we aim to study them theoretically by answering the question proposed by the chapter title. More specifically, we will identify what domain-independent heuristics we can or can not learn with them.

Sec. 4.1 and 4.2 encapsulate our main theoretical results for categorising the hierarchy of expressivity with MPNNs acting upon our graphs. The first section focuses on lower bounds, namely which domain-independent heuristics we are able to learn, while the second section focuses on upper bounds, namely which domain-independent heuristics we are not able to learn. Fig. 4.1 summarises the results from these two sections. Sec. 4.3 provides a discussion about the shortcomings and future directions of the results from the first two sections.

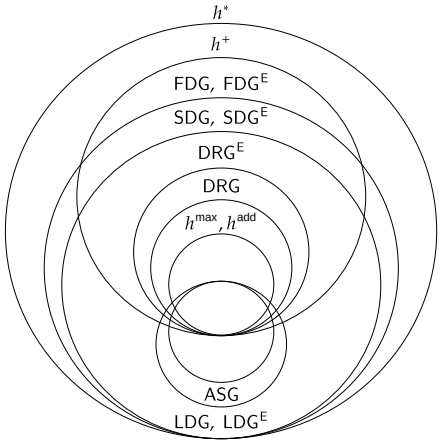



Figure 4.1: Hierarchy of expressivity with graphs from Ch. 3 combined with MPNNs.

## 4.1 Lower bounds

We will introduce some notation for classes of graphs that we have defined so far, with

- grounded graphs $\mathbf{G}_{\text{ground}} = \{\mathsf{DRG}, \mathsf{DRG}^{\mathsf{E}}, \mathsf{SDG}, \mathsf{SDG}^{\mathsf{E}}, \mathsf{FDG}, \mathsf{FDG}^{\mathsf{E}}\}$,
- lifted graphs $\mathbf{G}_{\text{lifted}} = \{\mathsf{ASG}, \mathsf{LDG}, \mathsf{LDG}^{\mathsf{E}}\}$, and
- all the defined graphs $\mathbf{G}_{\text{all}} = \mathbf{G}_{\text{ground}} \cup \mathbf{G}_{\text{lifted}}$.

As stated above, most of our theoretical results will be concerned with using the graph representations of planning problems in conjunction with only an MPNN. The reason for only considering MPNNs over other GRL models is because they are the most theoretically intuitive and the most computationally practical models for our setting.

In this subsection we focus on lower bounds, namely what domain-independent heuristics can we learn. The first result is that MPNNs can learn the $h^{\text{add}}$ and $h^{\max}$ heuristic on grounded graphs. The main idea of the proof is that MPNNs can imitate the dynamic programming approach to computing the heuristics by making use of approximation theorem for neural networks.

**Theorem 1** (MPNNs can learn $h^{\text{add}}$ and $h^{\max}$ on grounded graphs)**.** *Let $L, B \in \mathbb{N}$, $\mathcal{G} \in \mathbf{G}_{ground}$, $\varepsilon > 0$ and $h \in \{h^{\text{add}}, h^{\max}\}$. Then there exists a set of parameters $\boldsymbol{\Theta}$ for an MPNN $\mathcal{F}_{\boldsymbol{\Theta}}$ such that for all planning tasks $\Pi$, if Alg. 4 in the case of $h = h^{\max}$ or Alg. 3 when $h = h^{\text{add}}$ converges within $L$ iterations for $\Pi$, and $h(s_0) \leq B$, then we have $|h(s_0) - \mathcal{F}_{\boldsymbol{\Theta}}(\mathcal{G}(\Pi))| < \varepsilon$.*

*Proof.* The main idea of the proof is that we can encode Alg. 4 if $h = h^{\max}$ or Alg. 3 if $h = h^{\text{add}}$ into the MPNN framework in Alg. 5 using a correct choice of continuous bounded functions and aggregation operators and using the approximation theorem to find parameters in order to achieve the desired function. We will assume unitary cost actions and note that the below proof can be generalised to account for general cost actions. We first deal with the case where $h = h^{\max}$ and $\mathcal{G} = \mathsf{DRG}$.

Let $x^{(u)}$ be the feature of node $u$. By definition of $\mathsf{DRG}$, we can define the features by $x_0^{(u)} = 1$ if $u$ corresponds to a proposition node, else $x_1^{(u)} = 1$ when $u$ corresponds to an action node $a$. Furthermore, $x_2^{(u)} = 1$ if $u$ is a goal proposition, and $x_3^{(u)} = 1$ if $u$ is a proposition in the initial state. Note that it is possible that $x_2^{(u)} = x_3^{(u)} = 1$ when a proposition is both a goal condition and in the initial state. If not mentioned, we have that $x_i^{(u)} = 0$ everywhere else.

Then we will construct a MPNN with $2L + 2$ layers. For the first layer we have an embedding layer which ignores neighbourhood nodes with $\square^{(0)} = \vec{0}$ and $\varphi^{(0)}(\mathbf{h}_u, \mathbf{h}_N) = f_{\text{emb}}(\mathbf{h}_u)$. Let $K$ be the finite set of possible node features in a $\mathsf{DRG}$ representation of a planning task. Then $f_{\text{emb}} : K \to \mathbb{R}^3$ is defined by

$$f_{\text{emb}}([1,0,0,0]^\top) = [B,0,1]^\top \tag{4.1}$$
$$f_{\text{emb}}([0,1,0,0]^\top) = [0,0,0]^\top \tag{4.2}$$
$$f_{\text{emb}}([1,0,1,0]^\top) = [B,1,1]^\top \tag{4.3}$$
$$f_{\text{emb}}([1,0,0,1]^\top) = [0,0,1]^\top \tag{4.4}$$
$$f_{\text{emb}}([1,0,1,1]^\top) = [0,0,1]^\top. \tag{4.5}$$

This first round of message passing updates corresponds to the initialisation step of the heuristic algorithm with $B$ representing infinity values. We also note that after applying $\square^{(0)}$ and $\varphi^{(0)}$ and throughout the remaining forward pass of the MPNN, node embeddings will have the form $[x_0, x_1, x_2]$ which encode information about their corresponding proposition or action during the execution of the $h^{\max}$ algorithm where

- $x_0$ corresponds to the intermediate $h$ values computed in the $h^{\max}$ algorithm,
- $x_1$ signifies whether the node corresponds to a goal node, and
- $x_2$ determines if the node is a proposition or action node.

The next $2L$ layers use the component wise max aggregation function $\square = \max$ and alternates between setting $\varphi^{(l)}(\mathbf{h}_u, \mathbf{h}_N) = f_a([\mathbf{h}_u \,\|\, \mathbf{h}_N])$ and $\varphi^{(l+1)}(\mathbf{h}_u, \mathbf{h}_N) = f_p([\mathbf{h}_u \,\|\, \mathbf{h}_N])$ where $f_a : \mathbb{R}^6 \to \mathbb{R}^3$ and $f_p : \mathbb{R}^6 \to \mathbb{R}^3$ are defined by

$$f_a\left(\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ y_0 \\ y_1 \\ y_2 \end{bmatrix}\right) = \begin{bmatrix} x_0x_2 - (1-x_2)y_0 \\ x_1x_2 \\ x_2^2 \end{bmatrix}, \quad f_p\left(\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ y_0 \\ y_1 \\ y_2 \end{bmatrix}\right) = \begin{bmatrix} \min(x_0, -y_0+1)x_2 \\ x_1x_2 \\ x_2^2 \end{bmatrix}. \tag{4.6}$$

These functions correspond to the iterative updates of $h^{(l)}[a]$ and $h^{(l)}[p]$ in Alg. 4, recalling that $L$ is the number of iterations it takes for the algorithm converges. More specifically, suppose we have a node $u$ with embedding $\mathbf{h}_u = [x_0, x_1, x_2]$ and aggregated embedding from its neighbours $\mathbf{h}_N = [y_0, y_1, y_2]$. Then we have two cases.

- If $x_2 = 0$, indicating that the node $u$ corresponds to a n action, then we get

$$f_a([\mathbf{h}_u \,\|\, \mathbf{h}_N]) = [-y_0, 0, 0] \tag{4.7}$$
$$f_p([\mathbf{h}_u \,\|\, \mathbf{h}_N]) = [0,0,0]. \tag{4.8}$$

  Eq. 4.7 corresponds to Line 5 in Alg. 4 where $-y_0$ contains the negative of $h[a]$. We take the negative since we are restricted to using max aggregators only[1] which

[1] As min aggregators conflict with ReLU activation functions commonly seen in neural networks.

in turn means we require taking maximums of negatives in order to mimic the minimum aggregator later in Line 7 of the same algorithm. Eq. 4.8 corresponds to Line 7 but since this line only affects propositions and $h[a]$ values do not need to be stored after execution of this line, we set $\mathbf{h}_u$ to zero.

- If $x_2 = 1$, indicating that the node $u$ corresponds to a proposition, then we get

$$f_a([\mathbf{h}_u \Big\| \mathbf{h}_N]) = [x_0, x_1, x_2] \tag{4.9}$$

$$f_p([\mathbf{h}_u \Big\| \mathbf{h}_N]) = [\min(x_0, -y_0 + 1), x_1, x_2]. \tag{4.10}$$

We recall $f_a$ corresponds to Line 5 which only affects $h[a]$ values. Given that we require storing $h[p]$ values throughout the whole algorithm, $f_a$ acts as the identity function on $\mathbf{h}_u$ for proposition nodes as seen in Eq. 4.9. This is in contrast to $f_p$ which acts as the zero function on $\mathbf{h}_u$ for action nodes. Eq. 4.10 corresponds to Line 7 where $-y_0$ is equivalent to the $\min_{a \in A, p \in \text{add}(a)} h[a] = \max_{a \in A, p \in \text{add}(a)} -h[a]$ term by definition of DRG, $\square$ and $f_a$ acting on action node embeddings.

We append a final layer to the network where we ignore neighbourhood nodes with $\square^{(2L+1)} = \vec{0}$ and $\varphi^{(2L+1)}([x_0, x_1, x_2]^\top, \mathbf{h}_N) = x_0 x_1$. In combination with a max readout function $\Phi$, this corresponds to computing the final heuristic value. The above encoding of Alg. 4 has also been experimentally verified to be correct.

In order to satisfy the neural network component of the MPNN, we replace the $\varphi^{(i)}$ for $i = 0, \ldots, 2L + 1$ with feedforward networks. Noting that we have finitely many layers we can choose small enough fractions of $\varepsilon$ for the universal approximation theorem for neural networks [Hornik et al., 1989, Cybenko, 1989] to approximate the continuous functions $\varphi^{(i)}$ whose domain is bounded in the ball of radius $B$ in order to achieve our result.

The encoding for $h^{\text{add}}$ is the same except we use a sum aggregator $\square = \sum$ and readout.

For the case of the other grounded graphs, we note that they capture at least the same amount of information as DRG such that we can find similar MPNN encodings to represent approximations of $h^{\max}$ and $h^{\text{add}}$ with the given assumptions. Furthermore, we note that the $h^{\max}$ and $h^{\text{add}}$ algorithm for FDR problems and hence FDG graph representations work in the obvious way by compiling FDR planning tasks into propositional STRIPS planning task by treating variable-value pairs in FDR problems as propositional facts. $\square$

We note that we can set $\varepsilon = 0.49$ and append a rounding module to the output of an MPNN to get perfect computations of $h^{\max}$ and $h^{\text{add}}$ for integer valued cost functions as usually is the case for planning. It is also possible to extend the theorem to STRIPS-HGN by using additional assumptions accounting for their non-permutation invariant aggregation function.

Whether this is a useful theorem in practice is questionable given that we can compute $h^{\max}$ and $h^{\text{add}}$ using known algorithms such as Alg. 4 and 3. Furthermore, the theorem does not say anything about generalisation outside what the model has been trained on, as is usually the case with approximation theorem results. However, we may expect that MPNNs can generalise $h^{\max}$ and $h^{\text{add}}$ more efficiently by considering ideas of algorithmic alignment for improving sample complexity [Xu et al., 2020]. We note that their main theorems are concerned with approximating nice polynomial functions which is not the case in our encoding as we require an additional min function in Eq. 4.6.

Using our result, we can also learn $h^m$ if we allow for a polynomial time transformation of our task. Given a propositional STRIPS planning problem $\Pi = \langle P, A, s_0, G \rangle$, Haslum defines a transformation $\Pi^m = \langle P^m, A^m, s_0^m, G^m \rangle$ whose propositions are subsets of $P$ of at most size $m$ with the property that $h^m(s_0)$ for $\Pi$ is equivalent to $h^{\max}(s_0^m)$ for $\Pi^m$ [Haslum, 2009, Thm. 5]. We can leverage this fact for the following statement.

**Corollary 1** (MPNNs can learn $h^m$ on grounded graphs for $\Pi^m$)**.** *Let $L, B \in \mathbb{N}$, $\mathcal{G} \in \mathbf{G}_{ground}$ and $\varepsilon > 0$. Then there exists a set of parameters $\mathbf{\Theta}$ for an MPNN $\mathcal{F}_{\mathbf{\Theta}}$ such that for all planning tasks $\Pi$, if Alg. 4 converges within $L$ iterations for $\Pi$, and $h(s_0) \leq B$, then we have $|h^m(s_0) - \mathcal{F}_{\mathbf{\Theta}}(\mathcal{G}(\Pi^m))| < \varepsilon$.*

*Proof.* Given $L, B \in \mathbb{N}$, $\mathcal{G} \in \mathbf{G}_{\text{ground}}$ and $\varepsilon > 0$, Thm. 1 gives us that there exists some $\mathbf{\Theta}$ such that $|h^{\max}(s_0^m) - \mathcal{F}_{\mathbf{\Theta}}(\mathcal{G}(\Pi^m))| < \varepsilon$ for all planning tasks $\Pi$ for which Alg. 4 converges with $L$ iterations on $\Pi^m$. Then the result is immediate since $h^{\max}(s_0^m) = h^m(s_0)$ [Haslum, 2009, Thm. 5]. □

## 4.2 Upper bounds

Next we move on to negative results by constructing planning tasks in order to show what heuristics we cannot learn with our graphs and vanilla MPNNs. The keen reader might have noticed that we did not mention lifted graphs in the previous section. This is because as we see in the following result, they cannot learn $h^{\text{add}}$ or $h^{\max}$.

**Theorem 2** (MPNNs cannot learn $h^{\text{add}}$ and $h^{\max}$ on lifted graphs)**.** *Given $\mathcal{G} \in \mathbf{G}_{lifted}$, there does not exist any parameters $\mathbf{\Theta}$ for an MPNN such that it can correctly compute $h^{\text{add}}$ or $h^{\max}$ on all planning problems represented by $\mathcal{G}$.*

*Proof.* Consider the two (delete free) lifted STRIPS problems $P_1 = \langle \mathcal{P}, \mathcal{O}, \mathcal{A}, s_0^{(1)}, G \rangle$ and $P_2 = \langle \mathcal{P}, \mathcal{O}, \mathcal{A}, s_0^{(2)}, G \rangle$ with $\mathcal{P} = \{Q(x_1, x_2), W(x_1, x_2)\}$, $\mathcal{O} = \{o_1, o_2\}$, $s_0^{(1)} = \{Q(o_1, o_2), Q(o_2, o_1)\}$, $s_0^{(2)} = \{Q(o_1, o_1), Q(o_2, o_2)\}$, $G = \{W(o_1, o_2), W(o_2, o_1)\}$ and one action schema $\mathcal{A} = \{a\}$ with $\Delta(a) = \{\delta_1, \delta_2\}$, $\text{pre}(a) = \{Q(\delta_1, \delta_2)\}$, $\text{add}(a) = \{W(\delta_1, \delta_2)\}$ and $\text{del}(a) = \emptyset$.

By definition $P_1$ can be solved with a plan consisting of $a(o_1, o_2)$ and $a(o_2, o_1)$ in either order and the corresponding heuristic values are $h^{\max}(s_0^{(1)}) = h^{\text{add}}(s_0^{(1)}) = 1$. On the

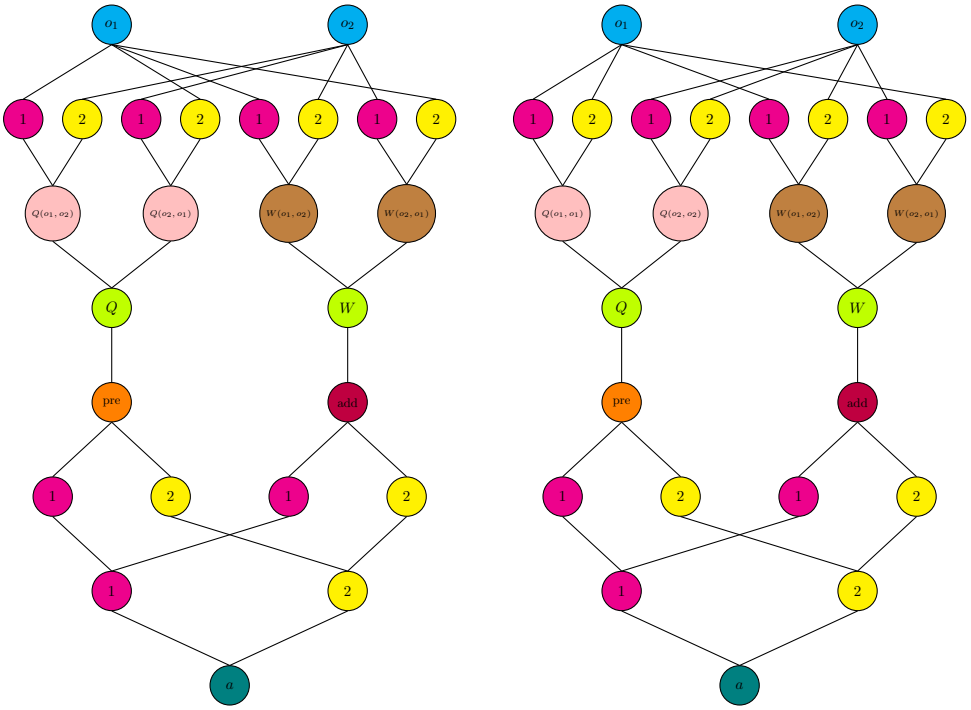



Figure 4.2: LDG of $P_1$ and $P_2$ (edges between objects and predicates omitted). Only colours are known to the WL algorithm, not the node descriptions in the figure. The only difference between the two graphs lies in the different edges between the top two layers of the graph. However, they are indistinguishable by the WL algorithm.

other hand $P_2$ is unsolvable in which case we have $h^{\max}(s_0^{(2)}) = h^{\text{add}}(s_0^{(2)}) = \infty$.

The graphs are indistinguishable by the WL algorithm where we colour nodes by mapping their features to the set of natural numbers for the LDG graphs, given that the set of possible node features is countable. Fig. 4.2 illustrates the graph representations for LDG. Then the result follows by the contrapositive of Lem. 1 as WL assigns the same output for both graphs, and hence any MPNN also assigns the same output. We can modify Lem. 1 to account for WL algorithms and MPNNs which deal with edge labels by compiling an edge coloured graph to a node coloured graph in the obvious way by replacing each edge with a node. Then this also gives us the result for $\mathsf{LDG}^{\mathsf{E}}$. The same pair of problems also applies for ASG as their ASG representations are also indistinguishable by the WL algorithm. □

The intuition of requiring symmetric goals for generating the counterexample is to construct a graph that is symmetric and regular enough for the pair to be indistinguishable by WL. Otherwise, we have asymmetries introduced by the action and predicate argument index features. Next, we show the importance of constructing graph representations

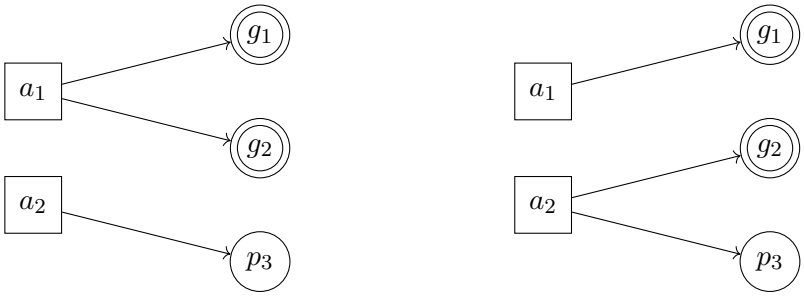



Figure 4.3: DRG of $P_1$ and $P_2$

without directed edges.

**Theorem 3** (MPNNs cannot learn $\{h^m(m > 1), h^{\text{SA}}, h^{\text{FF}}, h^+\}$ on DRGs)**.** *There does not exist any parameters* $\mathbf{\Theta}$ *for an MPNN (Alg. 5) such that it can correctly compute* $\{h^m(m > 1), h^{\text{SA}}, h^{\text{FF}}, h^+\}$ *on all planning problems represented by DRGs*[2]*.*

*Proof.* Consider the two (delete free) planning problems $P_1 = \langle P, A_1, s_0, G\rangle$ and $P_2 = \langle P, A_2, s_0, G\rangle$ with $P = \{g_1, g_2, p_3\}$, $G = \{g_1, g_2\}$, $s_0 = \emptyset$ and action sets $A_1 = \{a_1^{(1)}, a_2^{(1)}\}$, $A_2 = \{a_1^{(2)}, a_2^{(2)}\}$ where all the actions have empty preconditions and delete effects and

$$\text{add}(a_1^{(1)}) = \{g_1, g_2\}, \qquad \text{add}(a_1^{(2)}) = \{g_1\},$$
$$\text{add}(a_2^{(1)}) = \{p_3\}, \qquad \text{add}(a_2^{(2)}) = \{g_2, p_3\}.$$

We have that the minimum plan cost for $P_1$ is 1 by applying action $a_1^{(1)}$ whereas the minimum plan cost for $P_2$ is 2 as both actions need to be applied, as seen in Fig. 4.3. All the heuristics in the set $\{h^m(m > 1), h^{\text{SA}}, h^{\text{FF}}, h^+\}$ return 1 for $P_1$ and 2 for $P_2$.

Colour refinement assigns the same invariant to the DRG representations of $P_1$ and $P_2$ and thus by the contrapositive of Lem. 1, any MPNN assigns the same embedding to both graphs. □

As we can see, when we limit ourselves to forward flow of information we intuitively lose information of half of the structure of the graph. This can be seen in the ASG graph representation and also with the precondition nodes of STRIPS-HGN as seen in Fig. 3.1, although with STRIPS-HGN this is slightly alleviated with the recurrent global graph feature. It is easy to test and see that WL can distinguish these two planning graphs represented as $\text{DRG}^{\text{E}}$. However, our next result shows that MPNNs still cannot learn $h^+$ or any of the delete relaxation heuristics mentioned in Thm. 3 even if we have information flowing in both directions.

[2]Both $h^{\text{FF}}$ and $h^{\text{SA}}$ are only well defined with a specified tie breaking strategy but the proof of the theorem below works for any tie breaking strategy.

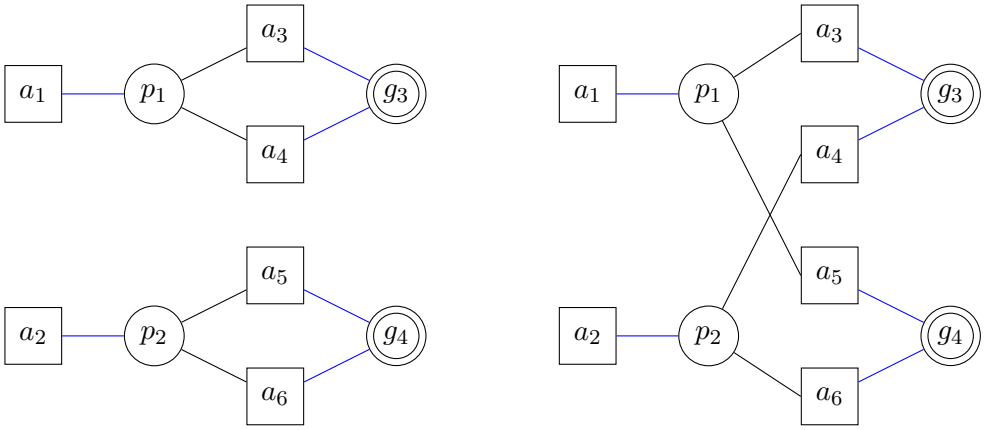



Figure 4.4: DRG$^{\mathsf{E}}$ of $P_1$ (left) and $P_2$ (right). Black edges indicate preconditions and blue edges indicate add effects.

**Theorem 4** (MPNNs cannot learn $\{h^m(m > 1), h^{\mathrm{SA}}, h^{\mathrm{FF}}, h^+\}$ on DRG$^{\mathsf{E}}$s)**.** *Let $h \in \{h^m(m > 1), h^{\mathrm{SA}}, h^{\mathrm{FF}}, h^+\}$. There does not exist any parameters $\boldsymbol{\Theta}$ for an MPNN (Alg. 5) such that it can correctly compute $h$ on all planning problems represented by DRG$^{\mathsf{E}}$s.*

*Proof.* Consider the two (delete free) planning problems $P_1 = \langle P, A_1, s_0, G\rangle$ and $P_2 = \langle P, A_2, s_0, G\rangle$ with $P = \{p_1, p_2, g_3, g_4\}$, $G = \{g_3, g_4\}$, $s_0 = \emptyset$ and action sets $A_1 = \{a_i^{(1)} \mid i = 1, \ldots, 6\}$, $A_2 = \{a_i^{(2)} \mid i = 1, \ldots, 6\}$ where actions have no delete effects and are defined by

$$
\begin{array}{llll}
\mathrm{pre}(a_1^{(1)}) = \emptyset, & \mathrm{add}(a_1^{(1)}) = \{p_1\}, & \mathrm{pre}(a_1^{(2)}) = \emptyset, & \mathrm{add}(a_1^{(2)}) = \{p_1\}, \\
\mathrm{pre}(a_2^{(1)}) = \emptyset, & \mathrm{add}(a_2^{(1)}) = \{p_2\}, & \mathrm{pre}(a_2^{(2)}) = \emptyset, & \mathrm{add}(a_2^{(2)}) = \{p_2\}, \\
\mathrm{pre}(a_3^{(1)}) = \{p_1\}, & \mathrm{add}(a_3^{(1)}) = \{g_3\}, & \mathrm{pre}(a_3^{(2)}) = \{p_1\}, & \mathrm{add}(a_3^{(2)}) = \{g_3\}, \\
\mathrm{pre}(a_4^{(1)}) = \{p_1\}, & \mathrm{add}(a_4^{(1)}) = \{g_3\}, & \mathrm{pre}(a_4^{(2)}) = \{p_1\}, & \mathrm{add}(a_4^{(2)}) = \{g_4\}, \\
\mathrm{pre}(a_5^{(1)}) = \{p_2\}, & \mathrm{add}(a_5^{(1)}) = \{g_4\}, & \mathrm{pre}(a_5^{(2)}) = \{p_2\}, & \mathrm{add}(a_5^{(2)}) = \{g_3\}, \\
\mathrm{pre}(a_6^{(1)}) = \{p_2\}, & \mathrm{add}(a_6^{(1)}) = \{g_4\}, & \mathrm{pre}(a_6^{(2)}) = \{p_2\}, & \mathrm{add}(a_6^{(2)}) = \{g_4\}.
\end{array}
$$

We have that the minimum plan cost for $P_1$ is 4 by applying actions $a_1^{(1)}, a_2^{(1)}, a_3^{(1)}, a_5^{(1)}$ whereas the minimum plan cost for $P_2$ is 3 with actions $a_1^{(1)}, a_3^{(1)}, a_5^{(1)}$. All the heuristics in the set $\{h^m(m > 1), h^{\mathrm{SA}}, h^{\mathrm{FF}}, h^+, h^*\}$ return 4 for $P_1$ and 3 for $P_2$.

Colour refinement modified to account for edge labels assigns the same invariant to the DRG$^{\mathsf{E}}$ representations of $P_1$ and $P_2$ as seen in Fig. 4.4. Thus by the contrapositive of Lem. 1 any MPNN assigns the same embedding to both graphs. □

It is also possible to extend the result to STRIPS-HGN with additional assumptions and variables to account for their aggregation function which limits the size of aggregated neighbour nodes. In order words, it is also true that STRIPS-HGN cannot learn $h^+$ regardless of the number of their parameters, which can be noticed by their restrictive information flow and the fact that global graph features do not help in distinguishing the graphs in Fig. 4.4.

**Corollary 2** (STRIPS-HGN cannot learn $\{h^m(m > 1), h^{\text{SA}}, h^{\text{FF}}, h^+\}$)**.** *Let* $h \in \{h^m(m > 1), h^{\text{SA}}, h^{\text{FF}}, h^+\}$. *There does not exist any parameters* $\boldsymbol{\Theta}$ *for STRIPS-HGN such that it can correctly compute* $h$.

The counterexample can be used to provide a strict upper bound on what MPNNs can learn in conjunction with any of our defined graphs with the following corollary.

**Corollary 3** (MPNNs cannot learn $h^*$)**.** *There does not exist any parameters* $\boldsymbol{\Theta}$ *for an MPNN such that it can correctly compute* $h^*$ *on all planning problems represented by any graph representation* $\mathcal{G} \in \mathbf{G}_{all}$.

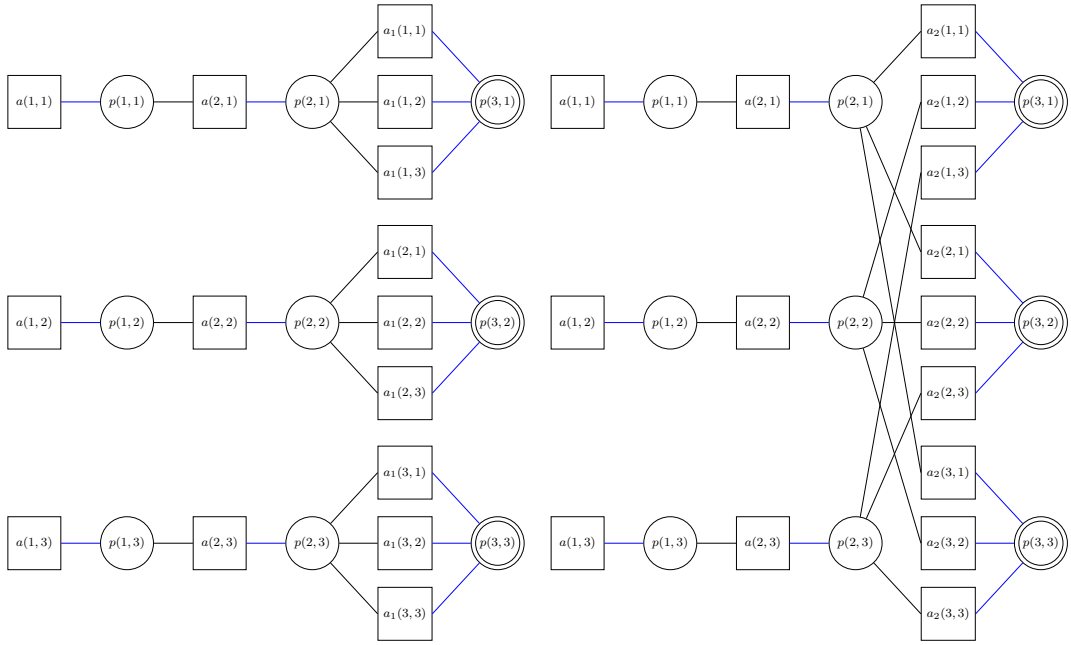
*Proof.* For $\mathcal{G} \in \{\mathsf{DRG}, \mathsf{DRG}^{\mathsf{E}}\}$ this follows directly from Thm. 4 since $h^* = h^+$ for the delete free problems in the proofs. The case is similar for $\mathcal{G} = \mathsf{SDG}^{\mathsf{E}}$ where the graph representation for the problems are the same as for $\mathsf{DRG}^{\mathsf{E}}$. For $\mathcal{G} = \mathsf{SDG}$, the graphs are similar except for the addition of auxiliary nodes but the graphs are still indistinguishable by colour refinement.

For $\mathsf{FDG}$ and $\mathsf{FDG}^{\mathsf{E}}$, if we use the naive compilation[3] of STRIPS to FDR by converting each proposition into a variable with a domain of size 2 representing whether the proposition is true or false in the current state, then we again get indistinguishable graphs with the WL algorithm.

For lifted graphs $\mathsf{ASG}$, $\mathsf{LDG}$, and $\mathsf{LDG}^{\mathsf{E}}$, we note that the problems specified earlier are grounded and the corresponding graphs do not leverage any additional information which makes them easier to distinguish. One can alternatively refer to Thm. 2. □

Even if it is not possible to learn the perfect heuristic, one may ask if it is possible to learn an approximation. We define an heuristic $h$ to be *c-absolutely approximately perfect* if $|h^*(s) - h(s)| < c$ for any state $s$ in any planning problem. An heuristic $h$ is *c-relatively approximately perfect* if $|1 - h(s)/h^*(s)| < c$ for any state $s$ in any planning problem. Note that we can define admissible versions of approximate heuristics by enforcing the admissibility criterion. This is done with $\varepsilon$-admissible heuristics [Pearl, 1984] which are defined to be heuristics satisfying $h(s) \leq (1 + \varepsilon)h^*(s)$ for all states $s$ which can be seen as one sided $\varepsilon$-relatively approximately perfect heuristics. We also note that these definitions of approximations are different than those which consist of a constant absolute error [Pohl, 1975, Helmert and Röger, 2008] parameterised by an integer $c \in \mathbb{N}$ and defined by $h^* - c := \max(h^* - c, 0)$. We do not consider these approximations since

[3] In fact, the more sophisticated translator used in Fast Downward provides the same compilation given that there are no mutually exclusive propositions here.




(a) $\mathsf{DRG}^{\mathsf{E}}$ of $P_1$ for $n = 3$. (b) $\mathsf{DRG}^{\mathsf{E}}$ of $P_2$ for $n = 3$.

Figure 4.5: $\mathsf{DRG}^{\mathsf{E}}$s of problems used in the proof of Thm. 5 where black edges indicate preconditions and blue edges indicate add effects.

in practice, if we were able to construct heuristic of this form and know $c$ in advance, then we can easily reconstruct $h^*$ by most states by adding $c$.

We see next that we can extend the counterexample graphs above to show that MPNNs cannot learn any approximation of the perfect heuristic. In other words, MPNNs can have arbitrarily bad predictions of the perfect heuristic.

**Theorem 5** (MPNNs cannot learn any approximation of $h^*$)**.** *There does not exist any parameters $\mathbf{\Theta}$ for an MPNN such that it can correctly compute c-absolutely approximately perfect or c-relatively approximately perfect heuristics for all $c \geq 0$ on all planning problems represented by any graph representation $\mathcal{G} \in \mathbf{G}_{all}$.*

*Proof.* Let us fix $n \in \mathbb{N}$ with $n > 2$. Then we will construct a pair of planning problems whose optimal plan costs are $2n - 1$ and $n^2$ respectively but are indistinguishable by MPNNs by any graph representations $\mathcal{G} \in \mathbf{G}_{\text{all}}$ of the problems. Thus, we can make our absolute and relative errors, given by $n^2 - 2n + 1$ and $\frac{n^2}{2n-1}$ respectively, arbitrary large.

Consider the two (delete free) planning problems given by $P_1 = \langle P, A_1, s_0, G\rangle$ and $P_2 = \langle O, A_2, s_0, G\rangle$ with $P = \{p(x, y) \mid x, y \in [n]\}$, $G = \{p(n, y) \mid y \in [n]\} \subset P$, $s_0 = \emptyset$ and actions $A_1 = \{a_1(y, z) \mid y, z \in [n]\} \cup A$ and $A_2 = \{a_2(y, z) \mid y, z \in [n]\} \cup A$ where $A = \{a(x, y) \mid x \in [n - 1], y \in [n]\}$. All actions have no delete effects and their preconditions

and add effects are given as follows

$$\begin{aligned}
\text{pre}(a(1,y)) &= \emptyset, & \text{add}(a(1,y)) &= \{p(1,y)\}, & \forall y &\in [n] \\
\text{pre}(a(x,y)) &= \{p(x-1,y)\}, & \text{add}(a(x,y)) &= \{p(x,y)\}, & \forall x &\in [2..n-1], y \in [n] \\
\text{pre}(a_1(y,z)) &= \{p(n-1,y)\}, & \text{add}(a_1(y,z)) &= \{p(n,y)\}, & \forall y, z &\in [n] \\
\text{pre}(a_2(y,z)) &= \{p(n-1,z)\}, & \text{add}(a_2(y,z)) &= \{p(n,y)\}, & \forall y, z &\in [n]
\end{aligned}$$

where we note that the case $n = 2$ is given in the proof of Thm. 4. We refer to Fig. 4.5 for the case of $n = 3$. An optimal plan for $P_1$ consists of executing all actions $a \in A$ and $a_1(y, 1)$ for $y \in [n]$. On the other hand, an optimal plan for $P_2$ consists only of executing $a(x, 1)$ for $x \in [n-1]$ followed by $a_2(y, 1)$ for all $y \in [n]$. Thus, the optimal plan costs for $P_1$ and $P_2$ are $n^2$ and $2n-1$ respectively. Furthermore as in the proof of Thm. 4, any graph representations of the pair of problems for any $n$ are indistinguishable by colour refinement and hence by MPNNs. □

## 4.3 Further discussion

In the previous two sections, we have classified what heuristic functions we can or cannot learn with MPNNs applied on our graphs. One might think the upper bounds we proved may be concerning as they state that we cannot learn any of the strong domain-independent heuristics accurately, and in the case of the lifted graphs, not even the $h^{\max}$ and $h^{\text{add}}$ heuristics. Thus the reader might ask, *why even bother learning domain-independent heuristics?* We can answer this in two ways.

### 4.3.1 A more refined hierarchy

The proofs only show the worst possible case where we have carefully constructed contrived planning task examples whose graph representations are indistinguishable by MPNNs. In practice, the graph representations of most planning tasks may be distinguished by MPNNs as will also observe in our first set of experiments later in Ch. 5. In other words, it is possible to learn the optimal heuristic on certain subclasses of all possible planning tasks. Given the scope of this thesis, it may be left for future work how we may identify classes of planning tasks where we can learn the optimal heuristic with MPNNs acting on the defined graph representations, and the probabilities on how often planning tasks lie in such classes. An analogy to this is hierarchical task network (HTN) planning where we introduce hierarchical actions in order to provide control knowledge to solve planning tasks more efficiently. The most general case is undecidable [Erol et al., 1996] but we may identify subclasses of problems with specific features and structures which are decidable and easier to solve [Alford et al., 2015, Chen and Bercher, 2021, 2022]. One may take an orthogonal approach and identify specific planning domains where we can learn the optimal heuristic [Staahlberg et al., 2022b]. We note that a subclass is more general than a planning domain, as the former may identify features that can generalise many planning domains.

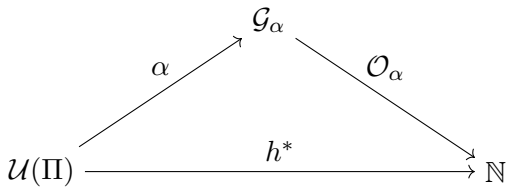



Figure 4.6: Instead of computing $h^*$, we can compute or learn $\mathcal{O}_\alpha$ instead.

### 4.3.2 More powerful GRL techniques

Our theory focuses on using MPNNs, but it is possible to rely on research progress on more powerful GRL models and techniques that allow us to learn $h^*$. To define this idea formally, let $\alpha \in \mathbf{G}_{\text{all}}$ be any graph representation, $\mathcal{U}(\Pi)$ denote the set of all possible planning tasks defined in the formalism associated with $\alpha$, and denote $\mathcal{G}_\alpha$ as the set of $\alpha$ graph representations from $\mathcal{U}(\Pi)$. Then our goal is to compute or learn an oracle function $\mathcal{O}_\alpha : \mathcal{G}_\alpha \to \mathbb{N}$ that corresponds to the perfect heuristic $h^*$ operating on all possible planning tasks.

For example, one may apply a universal theorem for MPNNs with random node initialisation in order to approximate $\mathcal{O}_\alpha$ with given bounds. Following the framework from Abboud et al., let $\varepsilon, \delta > 0$, $\mathcal{G}$ be a set of graphs and $f : \mathcal{G} \to \mathbb{R}$ be a function. Then we say that a randomised function $\mathbf{R}$ that associates every graph $G \in \mathcal{G}$ a random variable $\mathbf{R}(G)$ is an $(\varepsilon, \delta)$-*approximation of* $f$ if for all $G \in \mathcal{G}$, $\Pr(|f(G) - \mathbf{X}(G)| \leq \varepsilon) \geq 1 - \delta$. Next, given $n \in \mathbb{N}$ and $a \in \mathbf{G}_{\text{all}}$, denote $\mathcal{G}_\alpha^{\leq n}$ as the subset of $\mathcal{G}_\alpha$ with at most $n$ nodes.

Then directly applying the main theorem from the paper, we can derive the following powerful fact which states that we can approximate the optimal heuristic within a given confidence interval using MPNNs and partial random node initialisation (RNI) which consists of concatenating a random feature vector of any size to each node embedding of the graph. If we set $\varepsilon = 0.49$ and round the output of our MPNNs to the nearest integer, we can focus our attention entirely achieving perfect estimates under a certain probability. The main constraint is that the size of the model parameters scale with a specified number of arguments. If we fix the confidence $\delta$, the number of nodes $n$ of the graphs, and the maximum $H$ of computing $h^*$, then our embedding dimensions are given by $O(n^2\delta^{-1}H)$. If we relax the assumption that we can compute any $h^*$, the dimension is given by $O(n^2\delta^{-1}2^{\binom{n}{2}})$.

**Theorem 6** (MPNNs with partial RNI can approximate $h^*$)**.** *Let $\varepsilon, \delta > 0$, $n, H \in \mathbb{N}$ and $a \in \mathbf{G}_{all}$. Then for all $\varepsilon, \delta > 0$, there exists a set of parameters for an MPNN with partial RNI that $(\varepsilon, \delta)$-approximates $\mathcal{O}_\alpha : \mathcal{G}_\alpha^{\leq n} \to \mathbb{N}$ restricted to graphs $G \in \mathcal{G}_\alpha^{\leq n}$ where $\mathcal{O}_\alpha(G) \leq H$.*

*Proof sketch.* The proof follows the proof of Thm. 1 from [Abboud et al., 2021]. The main modifications we have to make to account for our graphs with node features are to

Lem. A.1. and A.3. from the appendix [Abboud et al., 2020]. In fact, the modification of the proofs apply to any graphs with a finite set of node features as suggested by the authors. The modification of Lem. A.3. involves additional encoding of our finite number of node feature vectors to encode individualised graphs with finite number of node features. We note that all our graph representations $\alpha \in \mathbf{G}_{\text{all}}$ have a finite number of node feature configurations when their size is bounded by $n$. The size limit prevents $\mathsf{LDG}$ and $\mathsf{LDG}^{\mathsf{E}}$ from having infinitely many node features due to the injective PE function. The proof of Lem. A.1. requires modifying the MPNN to preserve the initial node features after the first local message passing layer. □

The result may appear very appealing as the only overhead for approximating $h^*$ is in the additional model parameters given that random node initialisation can be done in constant time. However, its practicality is limited by the absence of any results concerning whether deep learning optimisers can yield parameters which find the correct MPNN parameters, and generalisability beyond the training set. We will see that this is the case when we apply RNI with MPNNs that we overfit and are able to achieve significant predictive performance boost on the training set at the cost of performance on unseen data.

There also exist many other feature augmentation techniques with empirical success for improving the expressivity and generalisability of MPNNs as we also observe in our first set of experiments in Ch. 5. One may also consider trying out more advanced GRL models over MPNNs such as the methods briefly surveyed in Sec. 2.3.3, although most of the methods come at a higher computational cost when considered in our setting.

Chapter 5

# Experiments 1: expressivity and generalisability

In Ch. 3 we defined new graph representations of planning problems for use with graph representation learning models as learned heuristic functions. In Ch. 4 we identified some lower and upper bounds in response to the question *what can we learn?* In this chapter, we conduct our first set of experiments in order to answer the following questions as to continue the conversation from our theoretical study, and discussion in Sec. 4.3:

**Does our theory match reality?**

We constructed novel graph representations of planning problems with an aim to improve on existing graphs with the goal of learning domain-independent heuristics. We also developed a hierarchy of expressivity of such graphs illustrated in Fig. 4.1 through our detailed theoretical discussion in the previous section. It is thus reasonable to perform some experiments to observe whether the results and ideas also hold in practice for a diverse set of planning tasks.

**What are the empirical upper bounds?**

Thm. 3 and Thm. 5 show that we cannot learn any approximation of the perfect heuristic accurately. However, the proofs of the theorems consider the worst case scenarios of contrived planning tasks and we suggested in Sec. 4.3.1 to relax our question and ask for what classes of planning problems can we still learn a good approximation of the perfect heuristic. Our experiments show that it is still possible to learn approximations of $h^*$ with our stronger graph representations and generalise to unseen data.

**Can more powerful GRL techniques aid us?**

In Sec. 4.3.2 we emphasised that our theoretical results only consider the most basic graph neural network models, namely MPNNs. It is possible to boost expressivity of our models with various GRL techniques from Sec. 2.3.3. In our experiments, we focus on techniques which accrue minimal computational overhead.

## 5.1 Setup

In a nutshell, our experiments perform the task of regression where the inputs are the graph representations of planning problems and the targets are the scalar value of the optimal cost to the goal. In this section we describe all the specifics of our experimental setup with all the details regarding the construction of the dataset, the training splits, model configurations and hyperparameters, the training pipeline and evaluation.

### 5.1.1 Dataset

In order to provide a comprehensive evaluation on the quality of our methods for the predicting heuristic values, we require a diverse set of planning tasks and their optimal costs from various domains. This is because a focus of our work is on learning domain-independent heuristics as opposed to domain-dependent heuristics with the motivation that we only have to train our model once and be able to apply it on any arbitrary planning task. Thus by providing a variety of domains, we may prevent our model from learning biases existent in domains as certain problems may have features dependent on their domain which makes learning easier.

Our dataset is the subset of all planning tasks from IPC 1998 to 2018 with unit costs where we are able to compute their optimal costs using the state-of-the-art `scorpion` planner[1] [Seipp et al., 2020]. We ran the solver on a cluster with Intel Xeon 3.2 GHz CPUs, a single core, 4GB memory and a timeout of 1800 seconds. We note that it is possible to generate more data by attempting to run other optimal planners on problems which `scorpion` cannot solve with the given resources. For each optimal plan of length $n$, we are able to generate $n+1$ initial states $s_0, s_1, \ldots, s_{n-1}, s_n$ from the optimal plan with target values $n, n-1, \ldots, 1, 0$[2] respectively. We note that it is unlikely that there is an efficient method to randomly generate good data with labels, as it has been shown that if we assume that $\mathsf{NP} \neq \mathsf{coNP}$[3], then there is no sample generator that can generate good labelled data for $\mathsf{NP}$-hard problems [Yehuda et al., 2020]. We refer to Sec. A.2 in the appendix for additional details about the dataset used.

Our training and test data is a partition of this dataset. We construct our training set to consist of 80% of the planning tasks whose target value is less than or equal to 32, and the test set to be the remaining 20% of the planning tasks with target value $h^* \leq 32$ and all the planning tasks with target value $h^* > 32$. We may further construct a validation set within our training set as we will describe later. Fig. 5.1 illustrates this partitioning. Performing this train and test split allows us to measure generalisability within and also outside our trained regression values.

[1] Available from https://github.com/jendrikseipp/scorpion

[2] Usually definitions of planning instances do not allow the initial state to be a goal. However this does not matter for the context of learning, as providing more samples with zero $h^*$ labels may help our models learn better.

[3] It is unknown whether these two complexity classes are equal or not.

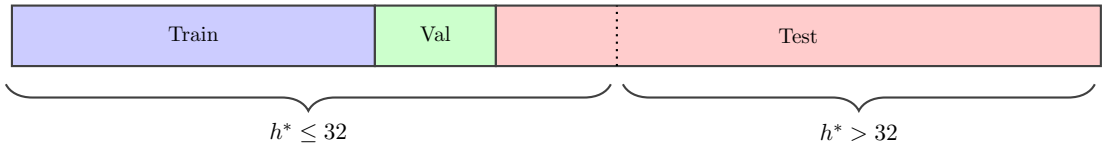



Figure 5.1: Training and test split of the IPC dataset. We further construct a validation set from the training set for scheduling the number of training epochs.

### 5.1.2 Model configurations

We construct one MPNN model for each of the 7 graph representations from {SDG, SDG$^{\mathsf{E}}$, FDG, FDG$^{\mathsf{E}}$, ASG, LDG, LDG$^{\mathsf{E}}$} as defined in Ch. 3. We omit the delete relaxation graphs since it does not make sense to train them to learn the perfect heuristic $h^*$ as they have no information about delete effects of actions. Thus, given a planning task with a domain, initial state and goal, we can make a prediction on its optimal heuristic value for search by converting the task into our choice of graph representation and passing it through our MPNN.

Representations without edge labels (SDG, FDG, ASG, LDG) use the following MPNN with update function

$$\mathbf{h}_u^{(t+1)} = \sigma\left(\mathbf{W}_0^{(t)}\mathbf{h}_u^{(t)} + \max_{v \in \mathcal{N}(u)} \mathbf{W}_1^{(t)}\mathbf{h}_v^{(t)}\right) \tag{5.1}$$

where the max is performed component wise over neighbour node representations. The motivation for the max aggregator arises from [Velickovic et al., 2020] which empirically shows its success in learning to imitate graph algorithms. This is also the case as in learning value functions by [Staahlberg et al., 2022b]. Furthermore, our informal experiments with using the sum and max aggregator show that the max aggregator is more robust to train and generalise despite losing information [Xu et al., 2019].

Representations with edge labels (SDG$^{\mathsf{E}}$, FDG$^{\mathsf{E}}$, LDG$^{\mathsf{E}}$) on the other hand used the relational graph convolutional networks (RGCN) [Schlichtkrull et al., 2018] modified with a max aggregator. Given a graph representation with edge labels $R$, its update function is given by

$$\mathbf{h}_u^{(t+1)} = \sigma\left(\mathbf{W}_0^{(t)}\mathbf{h}_u^{(t)} + \sum_{r \in R} \max_{v \in \mathcal{N}_r(u)} \mathbf{W}_r^{(t)}\mathbf{h}_v^{(t)}\right) \tag{5.2}$$

where $\mathcal{N}_r(u) = \{v \in \mathcal{N}(u) \mid \mathbf{E}((u,v)) = r\}$ denotes the neighbours of a node induced by edge label $r$.

We set the hidden dimension to be 64 across all update layers, such that $\mathbf{W}_0, \mathbf{W}_1, \mathbf{W}_r \in \mathbb{R}^{64\times64}$. Additional computing resources and hyperparameter search may be able to provide better hidden dimension values depending on our task. Furthermore, both of the MPNN models we use consist of an initial linear embedding layer $\mathbf{W} \in \mathbb{R}^{64\times n}$

where $n$ is the dimension of node embeddings of our input graph representations, and a readout layer consisting of a mean pooling over all nodes in the graph and a two layer MLP with 64 hidden dimension and an output dimension of 1. We further use residual layers [He et al., 2015] to improve gradient updates during backpropagation in the training procedure.

### 5.1.3 Feature augmentations

We also experiment with different feature and model augmentations to provide some insight into GRL techniques which can help us in our goal of learning good approximations of $h^*$ tractably. Specifically, we try

- concatenating Laplacian Positional Encodings (LPE) [Belkin and Niyogi, 2003, Dwivedi et al., 2020] of sizes 1 and 4 to the original node features,
- concatenating Random Node Initialisation (RNI) [Abboud et al., 2021] sampled from a normal distribution of sizes 1 and 4 to the original node features and this is done at every iteration of training or evaluation,
- adding a Virtual Node (VN) or global graph feature through each message passing step [Hu et al., 2020], and
- using a sum readout (SUM) for generating graph embeddings as opposed to mean.

LPE features are known to boost both expressivity and generalisation capabilities of models on molecular datasets and graph property detection tasks. They are computed as eigenvectors of the graph Laplacian which encode additional information about the structure of the graph. Random node initialisation may be used to break symmetries in the graph to improve expressivity and comes with universal theorems as discussed in Ch. 4. One may alternatively view RNI as a method for injecting noise to the dataset which is a common procedure done in machine learning tasks. Virtual nodes with different update functions are also known to improve predictive performance on molecular and syntax tree graph datasets [Hu et al., 2020] and allow MPNNs to learn first order predicate logic formulas with two variables and counting quantifiers [Barceló et al., 2020], a stronger result than Lem. 1 which does not mention learning a classifier.

### 5.1.4 Training pipeline and hyperparameters

We perform the following training pipeline for each graph representation and corresponding MPNN model described in Sec. 5.1.2 5 times to account for variance in the training procedure. Each model has a fixed number of 16 independent message passing layers and hidden dimension of 64. A model is trained with the Adam optimiser [Kingma and Ba, 2015] with the default parameters described in the paper, batch size of 16 and initial learning rate of 0.001. Our loss function is the mean squared error loss without regularisation on the weights of the model. We schedule our learning rate by constructing a validation set using 25% of the training data (or equivalently 20% of the whole

dataset with target $\leq 32$) and reducing the learning rate by a factor of 10 if the loss on the validation set did not decrease in the last 10 epochs. Training is stopped when the learning rate becomes less than $10^{-5}$. The model with the best weighted training and validation loss computed by

$$\texttt{weighted\_loss} = (\texttt{train\_loss} + 2 \times \texttt{val\_loss})/3 \tag{5.3}$$

is saved, with the idea that we do not want to overfit to either the training or validation set. After training our models, we append a rounding module such that their outputs are integer and we can evaluate them using classification metrics. Due to the costly training time and number of model configurations (graph representations + feature augmentations), we do not perform any hyperparameter search for the described parameters.

## 5.2 Results

We recall that we have partitioned our dataset into four sections as described in Fig. 5.1. We will evaluate our models on three of the sections: (1) the training set, (2) the test set with $h^* \leq 32$ and (3) the test set with $h^* > 32$. The motivation for evaluating on set (1) is to provide additional insight into the expressivity of our models for learning domain-independent heuristics. More specifically, we aim to empirically measure how good of an approximation of $h^*$ we can achieve in practice. This is to fill in the gap between the lower and upper bounds we derived in Ch. 4 and to show that Thm. 3 and 5 are not as bad as they seem. By evaluating on set (2), we test in typical machine learning fashion how well our models generalise to data not seen from the training set but are still from a similar distribution. The evaluation on set (3) is done to test how well our models extrapolate to data from outside the training distribution and whether they are actually learning to compute $h^*$, or selecting certain features that may help them associate graphs with their corresponding label.

### 5.2.1 Expressivity

We refer to the first major row of Tab. 5.1 for macro F1 scores[4] on set (1) with various graph representation and feature augmentation configurations. We observe that with no feature augmentations, the grounded graphs with edge labels ($\mathsf{FDG}^{\mathsf{E}}$ and $\mathsf{SDG}^{\mathsf{E}}$) perform best, followed by $\mathsf{FDG}$ and then $\mathsf{SDG}$, recalling that $\mathsf{SDG}$ has directed edges while $\mathsf{FDG}$ has undirected edges. The three worst performing graphs are the lifted graphs which encode planning tasks more compactly and thus are more difficult to learn with. From the lifted graphs, $\mathsf{LDG}^{\mathsf{E}}$ performs best, followed by $\mathsf{LDG}$ and then $\mathsf{ASG}$ with significantly

[4]The F1 score is the harmonic mean of precision and recall for binary classification. The harmonic mean punishes extreme values of precision and recall. The macro F1 score for multiclass classification is the average of F1 scores over each class, integer $h^*$ values in our case. The average is taken to account for class imbalance.

Table 5.1: Mean and standard deviation of macro F1 scores (scaled betweeen 0 and 100) for different configurations of graph representations and feature augmentations on subsets of the training and testing datasets. Cells are shaded blue if the score is greater than 50.0, with higher intensities for higher values, and shaded red if the score is less than 50.0, with higher intensities for lower values.

| Train | Grounded | | | | Lifted | | |
|---|---|---|---|---|---|---|---|
| Augmentation | SDG | SDG$^{E}$ | FDG | FDG$^{E}$ | ASG | LDG | LDG$^{E}$ |
| none | 54.1 ± 8.5 | 99.3 ± 0.3 | 90.5 ± 4.1 | 98.6 ± 0.3 | 16.5 ± 3.0 | 47.1 ± 4.1 | 50.4 ± 2.3 |
| LPE1 | 73.8 ± 1.5 | 99.8 ± 0.2 | 93.1 ± 2.8 | 98.8 ± 0.1 | 17.2 ± 1.4 | 43.2 ± 4.8 | 50.7 ± 3.3 |
| LPE4 | 80.0±13.1 | 99.9 ± 0.1 | 96.3 ± 3.4 | 99.9 ± 0.1 | 15.6 ± 2.6 | 39.0 ± 2.7 | 46.9 ± 3.3 |
| RNI1 | 76.9 ± 3.3 | 100.0±0.0 | 97.5 ± 2.2 | 99.2 ± 0.4 | 15.6 ± 3.2 | 65.3±11.3 | 88.4±11.5 |
| RNI4 | 82.4 ± 6.0 | 99.9 ± 0.2 | 99.7 ± 0.2 | 100.0±0.1 | 58.3±22.1 | 65.2±22.5 | 98.7 ± 2.1 |
| VN | 72.7 ± 3.9 | 99.2 ± 1.5 | 95.6 ± 2.2 | 98.8 ± 0.1 | 35.7 ± 1.4 | 50.5 ± 6.2 | 51.6 ± 3.1 |
| SUM | 35.3 ± 6.9 | 3.4 ± 1.6 | 30.0±29.4 | 36.4±23.8 | 14.3 ± 4.3 | 32.2 ± 8.3 | 41.8 ± 3.8 |
| Test ($h^* \leq 32$) | Grounded | | | | Lifted | | |
| Augmentation | SDG | SDG$^{E}$ | FDG | FDG$^{E}$ | ASG | LDG | LDG$^{E}$ |
| none | 37.0 ± 5.4 | 69.4 ± 2.5 | 65.1 ± 3.9 | 71.8 ± 2.7 | 16.6 ± 2.9 | 33.3 ± 2.0 | 35.8 ± 0.6 |
| LPE1 | 39.2 ± 1.9 | 62.7 ± 2.6 | 51.6 ± 4.0 | 72.4 ± 2.9 | 16.7 ± 1.1 | 32.2 ± 1.9 | 33.8 ± 1.5 |
| LPE4 | 33.2 ± 2.6 | 49.8 ± 1.6 | 40.3 ± 2.8 | 61.7 ± 4.2 | 13.7 ± 1.9 | 29.4 ± 0.8 | 29.5 ± 1.3 |
| RNI1 | 31.1 ± 0.6 | 55.1 ± 7.4 | 40.8 ± 9.0 | 55.6 ± 6.7 | 12.8 ± 1.7 | 18.0 ± 1.6 | 18.9 ± 0.9 |
| RNI4 | 28.1 ± 1.4 | 47.6 ± 8.3 | 36.4 ± 3.3 | 52.0 ± 3.8 | 10.1 ± 1.8 | 17.5 ± 1.1 | 18.7 ± 1.4 |
| VN | 48.8 ± 2.5 | 71.9 ± 1.8 | 70.9 ± 1.9 | 77.4 ± 1.5 | 31.8 ± 1.0 | 36.3 ± 1.9 | 37.2 ± 1.2 |
| SUM | 32.1 ± 4.6 | 3.6 ± 2.0 | 25.0±23.5 | 31.4±18.9 | 13.6 ± 3.6 | 27.4 ± 5.9 | 32.7 ± 2.2 |
| Test ($h^* > 32$) | Grounded | | | | Lifted | | |
| Augmentation | SDG | SDG$^{E}$ | FDG | FDG$^{E}$ | ASG | LDG | LDG$^{E}$ |
| none | 0.7 ± 0.1 | 1.6 ± 0.3 | 2.6 ± 0.7 | 2.7 ± 0.4 | 0.6 ± 0.4 | 1.3 ± 0.8 | 0.8 ± 0.2 |
| LPE1 | 1.3 ± 0.5 | 1.8 ± 0.6 | 1.6 ± 0.8 | 2.4 ± 0.3 | 0.9 ± 0.9 | 1.0 ± 0.5 | 1.0 ± 0.3 |
| LPE4 | 1.3 ± 0.1 | 1.1 ± 0.1 | 0.7 ± 0.1 | 1.6 ± 0.4 | 0.3 ± 0.3 | 1.6 ± 0.5 | 1.1 ± 0.2 |
| RNI1 | 0.5 ± 0.1 | 1.1 ± 0.4 | 0.7 ± 0.6 | 1.4 ± 0.4 | 0.6 ± 0.5 | 0.4 ± 0.1 | 0.4 ± 0.1 |
| RNI4 | 0.4 ± 0.1 | 1.0 ± 1.0 | 0.3 ± 0.1 | 1.1 ± 0.3 | 0.4 ± 0.4 | 0.3 ± 0.0 | 0.2 ± 0.0 |
| VN | 0.3 ± 0.1 | 1.4 ± 0.3 | 3.0 ± 2.3 | 1.9 ± 0.5 | 0.1 ± 0.0 | 0.4 ± 0.1 | 0.3 ± 0.0 |
| SUM | 13.2±13.6 | 0.3 ± 0.6 | 11.8±14.0 | 7.3 ± 3.0 | 0.3 ± 0.1 | 4.8 ± 1.9 | 7.4 ± 7.1 |

worse performance. We also note that LDG$^{E}$ and LDG have performance close to the grounded SDG graph.

The ranking of the graph representations stays consistent with different feature augmentations. The one exception is the case where SDG$^{E}$ performs significantly worse with a sum readout. When we look into the logs, we see that the training is more unstable than usual and our scheduler decays the learning rate too quickly, resulting in very early stopping during training. It is possible that increasing the patience of our scheduler may allow SDG$^{E}$ to achieve much better performance akin to the other grounded graphs with sum readout.

With regards to feature augmentations, we see that concatenating node features with random features (RNI1 and RNI4) provides a large boost to expressivity in training

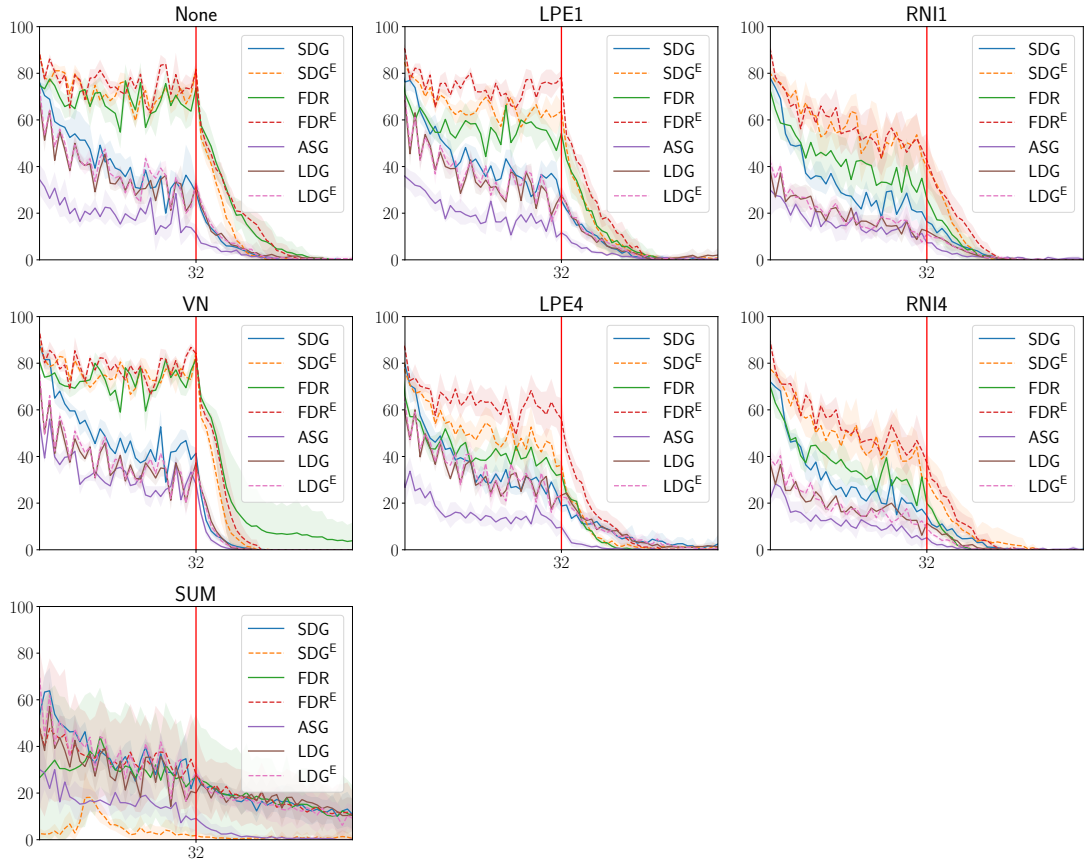



Figure 5.2: Mean and standard deviation of accuracy per target $h^*$ value for test sets with various graph representations and feature augmentations over 5 experiment repeats. The $y$-axes indicate accuracy (%) and the $x$-axes target $h^*$ value. The vertical red line indicates the interval of the $h^*$ values which the model was exposed to during training. Shaded regions indicate one standard deviation from the mean.

scores. The added noise makes the graphs easier to distinguish as discussed in Thm. 6. However its generalisation capabilities are limited by the fixed choice of hyperparameters, namely due to the short patience introduced by the learning rate scheduler. Informal experiments with greater patience provides much longer training times but better generalisation capabilities, similarly to results described in the original paper [Abboud et al., 2021]. Laplacian positional encodings (LPE1 and LPE4) also improve expressivity given that they inject graph structure information that MPNNs may not be able to detect by themselves. Increasing the size of RNI and LPE features generally leads to improved performance. Virtual nodes also aid with expressivity by allowing nodes to access information about the global graph feature during each layer. We notice that a sum readout results in significantly worse scores due to the difficulty of training with them. We note

that in theory sum readouts are more expressive than mean readouts [Xu et al., 2019] and lead to better generalisation as we will discuss below.

### 5.2.2 Generalisability

We refer to the second and third major rows of Tab. 5.1 for a quick numerical summary of generalisation performance, corresponding to sets (2) and (3). Fig. 5.2 provides plots of prediction accuracy conditioned on true $h^*$ values, and Fig. 5.3 provides confusion matrices to illustrate the actual predictions made by our models on the two test sets, with entries aggregated from all 5 repeats of the experiment.

The $i$-th row of a confusion matrix corresponds to samples with label $h^* = i$, while the $j$-th column correspond to samples which the model predicts $h^* = j$. The $i, j$-th entry of the matrix counts the number of samples with label $i$ which the model makes a prediction of $j$. Thus, higher counts on the diagonal of the matrix indicate better performance, counts on the bottom left of the matrix indicate underapproximations of $h^*$, and counts on the bottom right of the matrix indicate overapproximations.

#### Performance on seen class of labels ($h^* \leq 32$)

For generalisation performance on set (2), the ranking of graph representations remain the same as when we evaluate them on the training sets, with order from best to worst $\mathsf{FDG}^\mathsf{E}$, $\mathsf{SDG}^\mathsf{E}$, FDG, SDG, $\mathsf{LDG}^\mathsf{E}$, LDG, ASG. Again, we see that the main indicators of performance are whether the graph representations have edge labels, and whether they are grounded or not. We also note that FDR representations provide more information than grounded STRIPS representations as they explicitly encode mutex variables which MPNNs may not be able to learn. Furthermore, generalisation performance when we consider a sum readout is significantly worse when we measure with F1 or accuracy metrics and that their variance over several experiments is high which suggest that training them is more difficult. However, F1 and accuracy scores may not be a useful metric for evaluating learned heuristic functions, as we see in Fig. 5.3 that almost all configurations learn a reasonable approximation of $h^*$ on set (2) as the confusion matrices exhibit straight lines down the diagonal for samples with $h^* \leq 32$. We note that the accuracy of the approximations are given by the intensities of the lines where higher intensities correspond with higher confidence of our predictions.

#### Performance on unseen class of labels ($h^* > 32$)

Now we look at whether our models can extrapolate to samples in set (3). By referring to Fig. 5.2, we see that most of our models witness a sharp decline in performance as we increase the true labels of the samples. The exceptions to this are an instance of an MPNN with FDG and a virtual node, and the mean of the models with a sum readout which are able to extrapolate with a lower decline in performance as we increase the true labels. To illustrate the outlier FDG model, we refer the reader to Fig. B.1 in the appendix where we plot the maximum accuracies over the 5 experiment repeats.

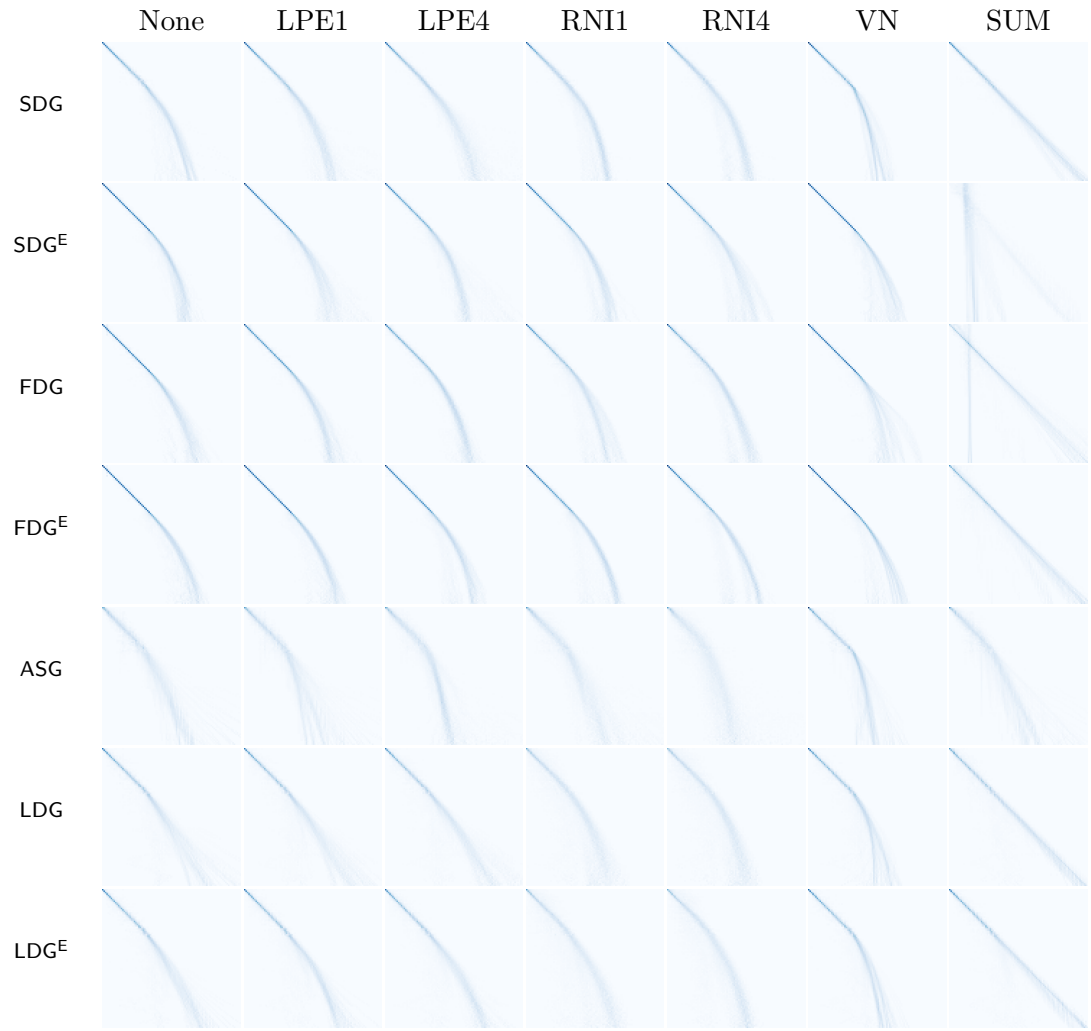



Figure 5.3: Confusion matrices of predicted and true heuristic values with various model configurations. Training was done on data with target heuristic value $h^* \leq 32$ (one third of the way down the rows and columns of the matrices). The $y$-axes correspond to the true label, and the $x$-axes the predicated label. Both axes are in the range $[0, 96]$ where values increase down the $y$-axes and increase from left to right of the $x$-axes.

The reasoning for steeper decline in performance of mean readout models lose information during the average of node features in its readout which tend to cause overfitting to the seen labels. Taking averages usually also lead to underestimating values as illustrated by the following example. Consider two graph representations $G_1$ and $G_2$ of the same planning problem where $G_1$ is a subgraph of $G_2$. We have that the nodes that are in $G_2$ but not in $G_1$ are all redundant isolated nodes. A mean readout graph will assign different embeddings to both graphs as they take the average of the node features,

whereas a sum readout graph will be able to assign the same embeddings if it learns to assign zero features to the isolated nodes. This intuition is illustrated in Fig. 5.3 where the predictions of the mean readout models underestimate $h^*$ outside the $h^*$ it has seen during its training. On the other hand, sum readout models are able to extrapolate reasonably well, illustrated by the straight line down the diagonal.

Furthermore, we point out that edge labelled graphs tend to extrapolate worse than their non edge labelled counterparts in some cases. This may be attributed to their higher expressivity and weight parameters which cause them to overfit to the set of seen $h^*$ values, and suggests that a hyperparameter search may improve their generalisation performance.

Lastly, we recall that sum readout graphs are less robust during training as seen with their high variance over several experiments. When viewing the confusion matrices, we note that SDG and certain instances of FDG train poorly and is stopped by the learning rate scheduler prematurely. This is indicated by the vertical beams in the corresponding confusion matrices as the models predict a small range of values for all samples.

### 5.2.3 Discussion

We now return to answer the questions we asked at the beginning of the chapter in order to complete the empirical analysis of our graphs:

#### Does our theory match reality?

Our theory was concerned with studying the expressivity of our graph representations in combination with MPNNs by focusing on the question *what can we learn?* The results do not explicitly say anything about the expected performance and generalisation capabilities of our models after the process of training on a dataset through the optimisation of a loss function. However, we do note that the experimental results align well with the information proposed by our theorems. Namely, the hierarchy of expressivity in Fig. 4.1 is observed in the evaluation of the trained models on both the training and unseen data on the same range of $h^*$ labels. Since our theorems do not mention anything about generalisation, the results on the unseen data with unseen $h^*$ labels do not contradict the theory.

#### What are the empirical upper bounds?

The high evaluation scores of the grounded graphs $\mathsf{SDG}^{\mathsf{E}}$, FDG, and $\mathsf{FDG}^{\mathsf{E}}$ on the training set highlighted in Fig. 5.1 suggest that it is still possible to learn good approximators of $h^*$. Referring to the evaluation of the models on unseen data with $h^* \leq 32$ in Fig. 5.3, we also see that the approximations almost always differ from $h^*$ by 2. When we use a sum readout, the models extrapolate to $h^*$ values of up to at least 96 with reasonable approximation as indicated by the diagonal lines in the confusion matrices.

We do note that the empirical results of these sets of experiments have its limitations. More specifically, the dataset may not be a good representation of all possible planning tasks since it only contains problems which were able to be solved optimally under given resource constraints, and the models were all trained and evaluated using the same set of hyperparameters. The picture may differ if we allow for hyperparameter tuning of our models which could improve the results, or conversely if we solved more difficult planning tasks to use as training samples the results may look worse. Furthermore, with more computational resources we can perform domain-independent heuristic evaluation in which we train on a specific set of planning domains and evaluate on unseen domains. Instead, we used our computational resources to evaluate domain-independent heuristics for search directly later in Ch. 7.

**Can more powerful GRL techniques aid us?**

Our results suggest that more powerful GRL techniques may aid us in improving the expressivity of our models as highlighted with the improved performance on the training set. However, with the exception of virtual nodes, they do not aid us in generalisation to unseen data, suggesting that they are causing our models to overfit to features rather than learning how to compute $h^*$ for arbitrary planning instances. It could be worth examining more advanced general case GNNs from the literature although we note that their evaluation is typically done on such as molecules or social networks which may easier to identify features correlated with their target labels, in contrast to our graph representations of planning domains which require long range reasoning over graphs in order to learn how to compute $h^*$.

Chapter 6

# The GOOSE framework

In Ch. 3, we have constructed graph representations of planning tasks with which we can use to learn heuristics. Following from this, in Ch. 4 and 5 we theoretically and empirically studied the capabilities and limits of using our graphs alongside MPNNs as learned heuristic functions. In this chapter, we introduce the GOOSE framework in Sec. 6.1 and then describe practical optimisations we have to consider to make learned heuristic functions feasible for search in Sec. 6.2.

## 6.1 Learning and planning

Our general GOOSE framework for solving satisficing planning problems consists of two core components:

1. a *learner* which constructs graph representations of planning instances for use with a corresponding learning method which accepts graph inputs, and
2. a *planner* which uses our learner component as a heuristic function to guide a heuristic search algorithm.

Given a graph representation $\mathcal{G}$ chosen from Ch. 3 and a graph representation learning model such as an MPNN $\mathcal{F}_\Theta$ with parameters $\Theta$, we have a heuristic function $h(s) = \mathcal{F}_\Theta(\mathcal{G}(\Pi))$ where $\Pi = \langle S, A, s, G \rangle$ is the planning task associated with $s$. The graph representation should match the corresponding planning formalism it is defined over. The MPNN in $h$ can be trained such that we are learning either domain-dependent or domain-independent heuristics as discussed in Sec. 2.2.4.

Then we can use our learned heuristic function $h$ in any heuristic search algorithm specified by our planner component. We also note that the planner component should transform the planning problem into a formalism compatible with $\mathcal{G}$. In the remainder

of the chapter, we discuss modifications of existing heuristic search algorithms in order to better suit GOOSE.

## 6.2 Optimising heuristic evaluation

One of the main weaknesses of neural network methods for learning heuristic functions is their slow evaluation time. This sometimes outweighs the speedups gained from the increased informativeness of the learned heuristics and fewer expansions and evaluations during search. Although GPUs are often used to massively parallelise and speedup the training of neural networks and their evaluations on objects with large vector representations such as images, this is not the case for previous works on constructing neural networks to estimate heuristic functions. The main reason for this is that neural network GPU utilisation is low when evaluating on states sequentially and since there is a nontrivial cost of transferring data between the CPU and GPU, works [Shen et al., 2020, Toyer et al., 2020] have noted that evaluating neural networks with GPUs does not result in a speedup and may even slow down evaluation.

### 6.2.1 Background of GPU usage and parallelisation in search

A natural remedy to the slow evaluation is to batch as many evaluations with the GPU as seen with Batch Weighted A* Search (BWAS) [Agostinelli et al., 2019]. The authors used batched neural network heuristic evaluations for the Rubik's cube problem in Weighted A$^*$ by expanding the front $N$ nodes of the search queue at every iteration at once as opposed to 1. This idea is not new and was first studied for general best first search algorithms with $k$-best first search ($k$-BFS) [Felner et al., 2003]. Although this reduces the average time spent per heuristic evaluation, most of the evaluations may be wasted as it is possible that the next 2nd to $N$th nodes in the queue may never be expanded in the sequential version of the algorithm. To give an extreme example, with the perfect heuristic $h^*$, the number of expansions performed by GBFS is exactly equal to the length of the plan. Thus, increasing $N$ naively may not be the best use of parallelisation in search as there are costs in parallelisation such as the sequential cost of expanding the nodes and moving data to and from the GPU. Furthermore, we note that batching expansions produces outputs **different** to the corresponding sequential algorithm. Thus, the same batching method employed in A$^*$ with admissible heuristics no longer return plans with optimality guarantees.

Nevertheless, for an inconsistent heuristic it is possible to view $N$ as an *exploration* parameter, since a node further back in the queue may be useful but will not be expanded in sequential search for a while due to reliance on the heuristic to guide search. This in turn means we can view sequential search as a purely *exploitative* method in the exploration vs. exploration framework commonly discussed in RL but also leveraged in the search community [Xie et al., 2014, Lipovetzky and Geffner, 2017]. Thus instead of viewing batching as a method to speed up heuristic evaluation, we also can view it as a regulariser during search when we have an unreliable heuristic. This coincidentally better

benefits learned heuristics which do not have any theoretical guarantees and which also profit greatly from the speedup of parallelisation via GPUs in comparison to classical heuristic methods whose parallelisation is usually much more limited, namely by the number of available threads and cores.

However, for the sake of this thesis and constraints on computational resources and evaluation time, we focus on parallel optimisations for search algorithms which have the **same** expected output as the sequential versions of the algorithms. We note that our parallelisation methods for search are intended for speeding up evaluation of neural network heuristics by optimising 'useful' GPU utilisation as opposed to parallelisation for multi-core processors for which there is a much more extensive literature.

Multi-core processor parallelisation methods focus on distributing the search space over different processors and communicating information about local open and closed lists. A main bottleneck with such parallelisation methods is the need to communicate search information between processors. Abstractions can be used to aid in detecting duplicate states and reduce the number of synchronisation points [Zhou and Hansen, 2007, Burns et al., 2010]. Hashing nodes to decide which processor's closed list one should go to can help with load balancing and asynchronous communication [Evett et al., 1995, Kishimoto et al., 2013]. We do note that there exist parallel heuristic search algorithms using hand crafted domain-specific heuristics for single [Zhou and Zeng, 2015] and multiple [He et al., 2021] GPUs with several shared open lists.

### 6.2.2 Parallelised lazy search

We first provide a parallel algorithm for lazy search with a focus on maximising 'useful' GPU utilisation. As we recall from Sec. 2.2.1, lazy GBFS is similar to the canonical eager GBFS but with delayed heuristic evaluation: we evaluate the heuristic of a node when it is popped from the queue rather than at the moment it was generated. In turn, the priority value of a state in the queue is given by the heuristic value of its parent node rather than its own heuristic value as in eager GBFS.

One way to parallelise lazy search is to batch heuristic evaluation of nodes in the queue. For example, once we pop a node from the queue we have to compute its heuristic value, but we may also do so with a few other states at the front of the queue whose heuristic has not been evaluated yet. However, batching as many heuristic evaluations as we can may not contribute to any speedups since a state evaluated in the queue may never be popped from the queue. Fig. 6.1 illustrates this case. Given that there are costs to parallelisation such as synchronisation, converting states to graphs, and moving the corresponding graph data into the GPU, naive batching may result in slower performance than sequential evaluation. To help with notation, we thus define an evaluation to be *useful* if the state which was evaluated will eventually be popped from the queue and expanded, and *useless* otherwise. For example in the sequential case where the batch size is always 1, every evaluation is useful.

One method to alleviate this issue is to change the batch size in real time to maximise the

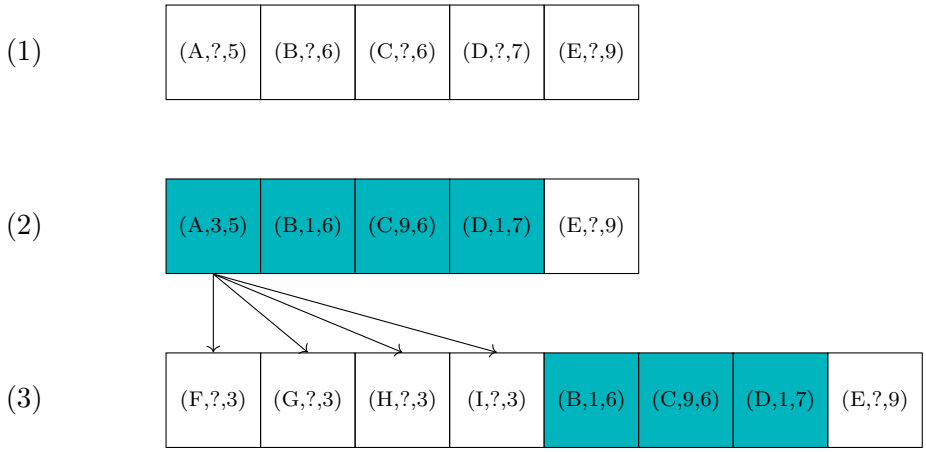



Figure 6.1: Greedy batched heuristic evaluation is not always useful. Each box represents a node in the queue of the form $(\alpha, h, h')$ where $\alpha$ is the name of the state, $h$ is the heuristic of the state and ? if not evaluated yet, and $h'$ is the heuristic of the parent node's state. The priority of the queue is defined by $h'$.
(1) The current state of the queue. (2) Batch evaluate the first 4 nodes in the queue. (3) The successors of node A are inserted into the front of the queue as their priority is given by $h(\mathrm{A}) = 3$ which is lower than the priority of any other node in the queue.

number of useful evaluations. The case illustrated in Fig. 6.1 does not necessarily happen all the time and one usually encounters plateaus or local minima during search depending on the heuristic. These are known as uninformative heuristic regions (UHRs) [Xie et al., 2014]. In this case, batching is extremely helpful as it speeds up the process of getting out of such UHRs as all evaluations are useful. Thus, we present Alg. 7 for detecting when increasing batch size does not significantly increase useless evaluations.

The main component for detecting when to increase the batch size is in Lines 10 to 19 which keeps count of the number of nodes popped from the queue since the last time we performed batched heuristic evaluation. Then we either increase or decrease the batch size depending on whether all previously batched heuristic evaluations were useful or not, which occurs when all evaluated states were popped from the queue before an unevaluated state gets popped. One assumption we make for this method is that the tie breaking method for ordering nodes with the same heuristic values in the priority queue is by considering the order which nodes enter the queue.

In practice, we may create and tune additional parameters to modify the rate of change of batch sizes, and relax the condition that all batched heuristic evaluations were useful to some percentage being useful.

**Algorithm 7:** Adaptive Batched Lazy GBFS

**Data:** Planning problem $\langle S, A, s_0, G\rangle$; heuristic function $h$; max batch size $U$.

```
1  OPEN ← ∅
2  s.closed ← ⊥,   ∀s ∈ S
3  s_0.h ← h(s_0)
4  OPEN.push(s_0, h(s_0))
5  n ← 1
6  iters ← 0
7  while OPEN ≠ ∅ do
8      s ← OPEN.popFront()
9      s.closed ← ⊤
10     if s.h = ? then
11         if iters ≥ n then
12             n ← n · 2
13         else
14             n ← ⌊n / 2⌋
15         n ← max(1, min(n, U))
16         iters ← 0
17         s'.h ← h(s') in parallel,  ∀s' ∈ {s} ∪ {front n nodes in OPEN with s.h = ?}
18     else
19         iters ← iters + 1
20     for a ∈ A do
21         t ← a(s)
22         if t ≠ ⊥ and t.closed = ⊥ then
23             if t ∈ G then
24                 return Extract plan from t
25             t.h ← ?
26             OPEN.push(t, s.h)
27 return No solution
```

### 6.2.3 Parallelised eager search

The most obvious method to perform parallelisation of heuristic evaluations for eager search is by parallelising the heuristic evaluation of successor nodes whenever we expand a node as outlined in Alg. 8 for GBFS. The main drawback to this method is that the parallelisation is bounded by the branching factor of our problem. For example in the $n^2 - 1$ sliding puzzle or a grid based path finding problem, we have at most 4 actions to perform at any state, meaning that the maximum theoretical speedup we gain from batching evaluations is 4.

**Algorithm 8:** Batched Eager GBFS

**Data:** Planning problem $\langle S, A, s_0, G \rangle$; heuristic function $h$.

```
1  OPEN ← ∅
2  s.closed ← ⊥,   ∀s ∈ S
3  OPEN.push(s_0, h(s_0))
4  while OPEN ≠ ∅ do
5      s ← OPEN.popFront()
6      s.closed ← ⊤
7      T ← {a(s) ≠ ⊥ | a ∈ A, a(s).closed = ⊥}
8      for t ∈ T do
9          if t ∈ G then
10             return Extract plan from t
11     t.h ← h(t) in parallel,   ∀t ∈ T
12     for t ∈ T do
13         OPEN.push(t, t.h)
14 return No solution
```

We could alleviate this issue by incorporating the adaptive batch size method used for batched lazy search to the eager case in combination with $k$-BFS. More specifically, we would keep a variable integer $k$ and in each iteration of the while loop, we would expand the first $k$ nodes in the queue and batch their successors' heuristic evaluation. Then we would increase or decrease $k$ by observing the local topology of the current search space in order to maximise useful evaluations. For example, if many nodes from the previous batch evaluation were added to the back of the queue then this is an indicator that we are in or approaching an UHR, in which case we would want to increase $k$. On the other hand if some nodes are added to the front of the queue, the heuristic may still seem to be informative so increasing $k$ may lead to useless evaluations.

Chapter 7

# Experiments 2: inference for search

In Ch. 3 we constructed a set of novel graph representations for planning tasks and evaluated the predictive capabilities of such graphs via MPNNs theoretically and empirically in Ch. 4 and 5 respectively. We then introduced the GOOSE framework and discussed intelligent methods for speeding up neural network heuristic evaluations during search in Ch. 6 with GPUs that go beyond naive methods discussed in the literature.

In this chapter we put together everything we have explored and perform a comprehensive set of experiments to evaluate how effective GOOSE is at learning to solve planning tasks quickly. We further perform a comprehensive evaluation of our models in comparison to classical heuristics in different learning settings and over a diverse set of domains, with results suggesting that our model is the state-of-the-art learner for planning in both domain-dependent and domain-independent learning settings.

Table 7.1: Summary of domains considered, objects whose number can vary, optimal plan cost if it can be computed in polynomial time or complexity for computing the optimal plan, and minimal and maximal branching factor during search.

| Domain | Objects | Opt. plan length/ Opt. complexity | Branching Min. | Branching Max. |
|---|---|---|---|---|
| BLOCKSWORLD | $b$ blocks | NP | 1 | $b+1$ |
| FERRY | $c$ cars, $l$ locations | $4c$ | $l-1$ | $c+l-1$ |
| GRIPPER | $b$ balls | $3b$ | 1 | $b+1$ |
| HANOI | $d$ discs | $2^d-1$ | 2 | 3 |
| $n$-PUZZLE | grid size $n$ | NP | 2 | 4 |
| SOKOBAN | $b$ boxes, $w$ walls, grid size $n$ | PSPACE | 1 | 4 |
| SPANNER | $s$ spanners, $n$ nuts, $l$ locations | $2n+l+1$ | 0 | $sn$ |
| VISITALL | grid size $n$ | $n^2-1$ | 2 | 4 |
| VISITSOME | grid size $n$ | NP | 2 | 4 |

## 7.1 Benchmark domains

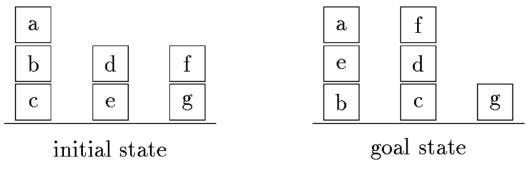
We begin the chapter by describing the benchmark domains we use in our experiments. It is important to understand the semantics of the domains in order to better understand how and why our models and the baselines perform in a certain way. We use the benchmarks used in the evaluation of STRIPS-HGN [Shen et al., 2020] which could be parsed by our planner with some additional benchmarks. Key information about the considered domains are summarised in Tab. 7.1.

### 7.1.1 Blocksworld

The Blocksworld domain consists of a set of blocks stacked to form one or more towers on top of a table and the objective of a planner is to stack and unstack blocks to achieve a target configuration of towers. There are many variants of the Blocksworld domain and we consider the canonical deterministic 4-operation Blocksworld domain where the possible moves for an agent is to

1. `stack` a block on top of another block,
2. `unstack` a block from another block,
3. `pick-up` a block from the table, or
4. `put-down` a block onto the table.

It is simple to solve any Blocksworld problem by unstacking and putting down all the blocks in the initial state onto the table and then picking up and stacking them in order to achieve the goal configuration. This provides us an upper bound on the optimal plan length as $4b$ where $b$ is the number of blocks in the instance where each block goes through a sequence of `unstack`, `put-down`, `pick-up` and `stack`. On the other hand, finding an optimal plan is NP-hard [Chenoweth, 1991, Gupta and Nau, 1991, 1992]. We refer to [Slaney and Thiébaux, 2001] for a comprehensive survey of the Blocksworld domain, including efficient algorithms for satisficing and optimal Blocksworld and generating[1] difficult Blocksworld instances.




| | | |
|---|---|---|
| 1. move a to table | 2. move b to table | 3. move d to c |
| 4. move e to b | 5. move a to e | 6. move f to d |

Figure 7.1: A Blocksworld instance with a description of the optimal plan, from [Slaney and Thiébaux, 2001]. The move action encapsulates some sequence of actions from {`unstack`, `put-down`, `pick-up`, `stack`}.

[1] The generator is available at https://gitlab.com/thiebaux/blocks-world-generator-and-planner

### 7.1.2 Ferry

Ferry is a simple transportation domain where a planner has to move $c$ cars each from their initial location to a specified location with a single ferry in a world containing $l$ locations. The possible moves for a planner to execute are

1. `sail` the ferry which may or may not carry a car from one location to another,
2. `board` a single car onto the ferry, and
3. `debark` a loaded car at the current location.

It is possible to see in general a tight upper bound on the optimal plan length of any Ferry instance is $4c$ since for each car the ferry has to `sail` to the location of the car, `board` it, `sail` to the goal location, and then `debark` the loaded car. The exact plan length varies with whether the cars are already at the goal location and if the ferry is initially next to a car.

### 7.1.3 Gripper

Gripper can be viewed as another transportation problem involving $n$ balls and 2 rooms where all the balls are located in Room A and the goal is to move all the balls into room B. The main difference to Ferry is that the number of locations is fixed and that we have two grippers meaning that we can pick up 2 objects before moving. The possible moves for a planner to execute are

1. `move` the agent from one room to another,
2. `pick` up a ball in the current room with one of the two grippers, and
3. `drop` a ball in the current room in one of the grippers.

The optimal plan length for an even number of balls is $3n-1$ and $3n$ for an odd number of balls.

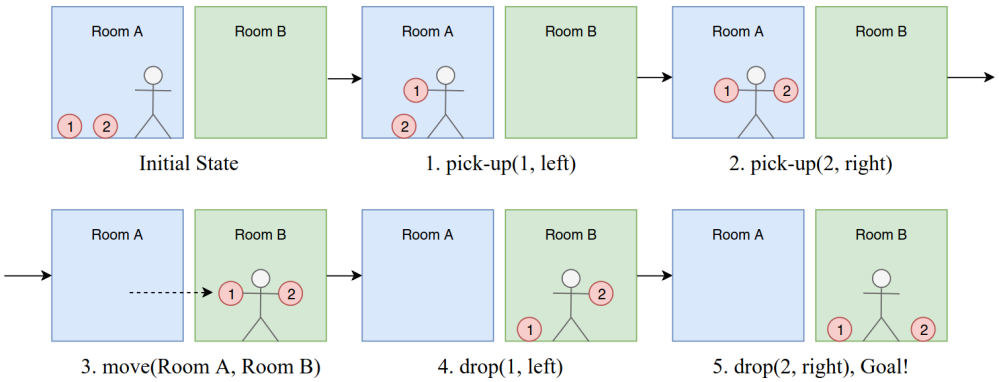



Figure 7.2: A gripper instance with two balls and a description of the optimal plan, from [Shen et al., 2020].

### 7.1.4 Hanoi

The Tower of Hanoi, or simply Hanoi, is a simple puzzle game consisting of three pegs and a set of $d$ discs all with different diameters. In the initial state, the discs are stacked onto the first peg with decreasing diameter size and the objective of the problem is to move the discs such that they are all stacked onto the last peg again with decreasing diameter size. However, in any move, we cannot have a disc stacked on another disc with a smaller diameter. The optimal plan for an instance with $d$ discs described in this way has length $2^d - 1$ and can be computed in a recursive manner. In practice, one can construct more than one problem for any fixed $d$ such that the difficulty of the problems does not grow too quickly (exponentially).

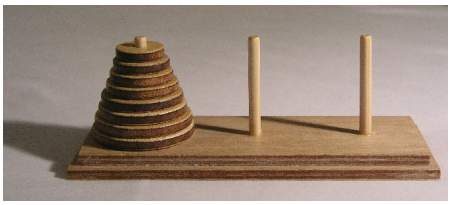


Figure 7.3: A real life Tower of Hanoi instance, from https://en.wikipedia.org/wiki/Tower_of_Hanoi.

### 7.1.5 $n$-puzzle

The 15 puzzle is another real life puzzle game consisting of 15 tiles on a 4 by 4 grid where the goal is to slide the tiles using the single empty space to achieve a certain configuration, usually with all the numbered tiles in order. We can generalise the puzzle by using an $n$ by $n$ grid and refer to it as the $(n^2 - 1)$-puzzle domain. It is NP-hard to solve optimally [Ratner and Warmuth, 1986] and an upper bound for the optimal plan is given by $O(n^3)$ [Parberry, 1995]. We note that the $n$-puzzle can be seen as a special case of undirected multi-agent pathfinding (MAPF) which itself is a special case of directed multi-agent pathfinding (diMAPF) which has recently been shown to be NP-complete [Nebel, 2022].

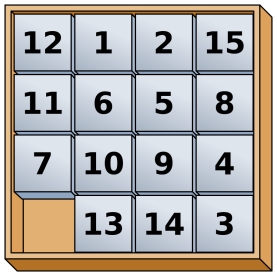



Figure 7.4: A 15 puzzle instance, from https://en.wikipedia.org/wiki/15_puzzle.

### 7.1.6 Sokoban

Sokoban is based on a video game where a player has to push $b$ boxes around a warehouse to their target locations. The warehouse is represented by an $n$ by $n$ grid and there are some walls representing untraversable regions. A player can either `move` one step in a cardinal direction to an unoccupied location or `push` a box one step, again to an unoccupied location. Since we cannot pull boxes, it is possible for Sokoban instances to have dead ends such as when a box gets pushed into a corner. The problem is PSPACE-complete [Culberson, 1997], even in the case where there are no walls [Hearn and Demaine, 2005].

### 7.1.7 Spanner

Spanner consists of a one way corridor with $l$ locations excluding the start (shed) and end (gate), littered with $s$ spanners on the ground and $n$ nuts at the gate. An agent is initially at the shed and has to pick up spanners to fix the nuts at the gate, where a spanner can only be used to fix one nut before it breaks. Thus instances are only solvable for $s \geq n$. The three possible types of actions a planner can perform are

1. `walk`, move the agent one step,
2. `pickup-spanner` which picks up one spanner, and
3. `tighten-nut` which fixes a nut but exhausts one spanner.

The domain was constructed for the learning track of the 2011 IPC and was constructed such that delete relaxation heuristics perform poorly as they do not account for the directed search space and breakable spanners. The optimal plan, if one exists, has length $2n + l + 1$, with $n$ `pickup-spanner` and `tighten-nut` actions, and $l + 1$ `walk` actions. The maximum branching factor is $sn$ in the case we picked up all the spanners and are beginning to fix the nuts. For large $s$ and $n$, this branching factor becomes a large bottleneck for eager search where we have to evaluate all $sn$ successor nodes even though they are all symmetric in the sense that it does not matter what spanner you use to fix which nut.

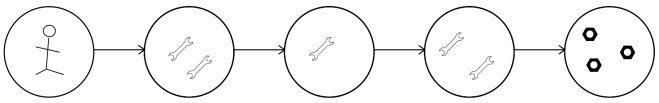


Figure 7.5: A spanner instance with 5 spanners, 3 nuts and 3 locations.

### 7.1.8 VisitAll

VisitAll is a domain in the classical tracks of IPC 2011 and 2014 which consists of an $n$ by $n$ grid and the goal of the problem is to visit all the locations of the grid, given some random starting point. VisitAll is another domain in which delete relaxation heuristics

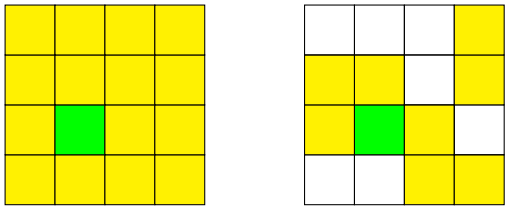


Figure 7.6: VisitAll (left) vs VisitSome (right). You have to either visit all or some of the goal locations marked in yellow, starting from some initial state marked in green.

perform poorly due to large uninformative heuristic regions (UHRs). The optimal plan cost is $n^2 - 1$ and is trivially solved by the goal count heuristic with GBFS.

### 7.1.9 VisitSome

We name VisitSome as the variant of VisitAll where we only have to visit a fraction of the locations on the grid. It is equally difficult for delete relaxation heuristics, and slightly harder for the goal count heuristic as it requires some searching due to goal locations not being adjacent to one another. One should note that this is a special case of the Hamiltonian path problem which is NP-hard to solve optimally [Garey and Johnson, 1979].

## 7.2 GOOSE setup

We recall that GOOSE consists of a *learner* component which constructs graph representations of planning instances for use with corresponding learning method which accepts graph inputs, and a *planner* which uses our learner component as a heuristic function to guide a heuristic search algorithm. We outline the implementation details of the two components for our experiments below.

### 7.2.1 Learner

We use the MPNN models described in Ch. 5 in conjunction with the graph representations from $\{\mathsf{SDG}, \mathsf{SDG}^{\mathsf{E}}, \mathsf{FDG}, \mathsf{FDG}^{\mathsf{E}}, \mathsf{LDG}, \mathsf{LDG}^{\mathsf{E}}\}$ from Ch. 3 as the first component of our GOOSE framework. We recall the update equations of the MPNNs for graph representations without and with edge labels respectively by

$$\mathbf{h}_u^{(t+1)} = \sigma\left(\mathbf{W}_0^{(t)}\mathbf{h}_u^{(t)} + \max_{v \in \mathcal{N}(u)} \mathbf{W}_1^{(t)}\mathbf{h}_v^{(t)}\right) \tag{5.1}$$

$$\mathbf{h}_u^{(t+1)} = \sigma\left(\mathbf{W}_0^{(t)}\mathbf{h}_u^{(t)} + \sum_{r \in R} \max_{v \in \mathcal{N}_r(u)} \mathbf{W}_r^{(t)}\mathbf{h}_v^{(t)}\right). \tag{5.2}$$

We also experiment with both mean and sum readouts but do not use feature augmentations. We describe our training setup later in Sec. 7.3.

### 7.2.2 Planner

The search algorithm is the second component of GOOSE. We use our trained MPNN models to provide heuristic estimates for both eager and lazy GBFS search in the C++ Powerlifted planner (PWL)[2] [Corrêa et al., 2020]. The models are implemented using the PyTorch Geometric 11.3 library [Fey and Lenssen, 2019] and called from the planner with `pybind11` [Jakob et al., 2017].

We leverage GPUs alongside the batched heuristic evaluation algorithms described in Alg. 8 and Alg. 7. We note that problems with higher branching factors for the eager case or heuristics with large UHRs for the lazy case are expected to see a greater speedup from utilising GPUs. All experiments are run on a cluster with single AMD EPYC 7282 2.8GHz CPU cores and single NVIDIA GeForce RTX 3090 GPU within a Singularity container with CUDA 11.6.

## 7.3 Experimental setup

Our goal is to construct learned heuristic functions to aid speed up search in unseen settings. We can evaluate our GOOSE framework in two dimensions of difficulty, noting that solving more difficult tasks allows greater usage of such models. The first dimension of difficulty is learning either domain-dependent and domain-independent heuristics as discussed in Sec. 2.2.4. The second dimension is evaluating on instances with the same range of sizes as seen in the training samples when considering domain-dependent heuristics, in comparison to evaluating on instances of arbitrarily large sizes. In the learning domain-independent heuristics setting, this amounts to difficulty varying in the sizes of the problems. Tab. 7.2 summarises the degrees of difficulty of various taxonomies of learning heuristics.

Table 7.2: Varying degrees of difficulty of learning for planning evaluation ranked from easiest to hardest.

| | | difficulty → | |
|---|---|---|---|
| | | Domain-dependent | Domain-independent |
| difficulty ↓ | Seen/small sizes | 1 | 3 |
| | Unseen/large sizes | 2 | 4 |

### Domain-dependent vs domain-independent training

The first dimension is by performing either domain-dependent or domain-independent training. Domain-dependent training consists of learning heuristic functions for a specific domain and testing on unseen instances from the same domain. Domain-independent

[2] Available at https://github.com/abcorrea/powerlifted

training consists of learning heuristic functions from a set of domains but testing on instances from a domain not seen in the training set. The former case is generally what is done in the learning for planning community and is also the norm for reinforcement learning as it is an easier problem to solve and methods in this direction are competitive in the number of expanded nodes with classical heuristics for some domains, although not so much the case with runtime. Domain-independent training is a difficult task as models have to learn about the semantics of planning problems instead of 'tricks' in specific domains in order to generalise effectively. To the best of our knowledge, STRIPS-HGN is the only model which was extensively evaluated with domain-independent training, but we note that the underlying model is limited by the a priori fixed size of action preconditions and effects.

A motivation to consider research in domain-independent training is that (1) compute and (hence) data is becoming cheaper and more accessible, and (2) there exist domain-independent training, termed domain adaptation, in other fields of learning such as computer vision. We note that methods and ideas from domain adaptation and zero shot learning may be applied for our setting but may require further thought due to the more abstract and unstructured setting of planning problems compared to images.

#### Seen/small vs unseen/large problem sizes

The second dimension, assuming domain-dependent training, is testing on the sizes of the problem. For example, with seen size evaluation, we train on small Blocksworld instances of up to 10 blocks and test on unseen Blocksworld instances again of only up to 10 blocks. The reinforcement learning analogy is learning specifically to play or solve board or video games such as Go [Silver et al., 2016] or Starcraft [Vinyals et al., 2019] really well. However, it may be the case that training data for large problems is difficult to achieve and it might be easier to get training data for smaller problems and then generalise to solve the larger instances. By being able to generalise to larger instances, we may also be able to handle unexpected cases not seen in the training data. As a concrete example, we again train on Blocksworld instances of up to 10 blocks but now test on unseen Blocksworld instances with more than 10 blocks.

### 7.3.1 Testing instances

The testing instances are different for the seen/small sizes and unseen/large sizes categories, described in Tab. 7.3 for each domain. We do not differentiate the instances based on whether we performed domain-dependent or domain-independent training.

### 7.3.2 Training pipeline and model selection

We want to train one MPNN model for each graph representation in order to evaluate with search. We will describe how we train a model for each of the 4 evaluation methods.

For each benchmark domain, we specify the training set described in Tab. 7.3 for

Table 7.3: Planning domains used for our evaluation with training, validation and test set splits. We note that the validation and test sets do not have any associated ground truth values such as an optimal plan or $h^*$. The symbol $*$ indicates overlap with training set, † indicates not all problems were solved from the description.

| Domain | Split | Instances | Description |
|---|---|---|---|
| Blocksworld | Train | 480 | $60 \times \{3, \ldots, 10$ blocks$\}$ |
| | Val | 3 | $3 \times \{11$ blocks$\}$ |
| | Test (small) | 40 | $5 \times \{3, \ldots, 10$ blocks$\}$ |
| | Test (large) | 90 | $5 \times \{15, 20, \ldots, 100$ blocks$\}$ |
| Ferry | Train | 810 | $10 \times \{2, \ldots, 10$ locations$\} \times \{2, \ldots, 10$ cars$\}$ |
| | Val | 3 | $3 \times \{(11, 11)$ (locations, cars)$\}$ |
| | Test (small) | 125 | $5 \times \{2, 4, \ldots, 8, 10$ locations$\} \times \{2, 4, \ldots, 8, 10$ cars$\}$ |
| | Test (large) | 90 | $5 \times \{(15, 15), (20, 20), \ldots, (100, 100)$ (locations, cars)$\}$ |
| Gripper | Train | 10 | $\{1, \ldots, 10$ balls$\}$ |
| | Val | 1 | $\{11$ balls$\}$ |
| | Test (small) | $10^*$ | $\{1, \ldots, 10$ balls$\}$ |
| | Test (large) | 18 | $\{15, 20, \ldots, 100$ balls$\}$ |
| Hanoi | Train | 8 | $\{3, \ldots, 10$ discs$\}$ |
| | Val | 1 | $\{11$ discs$\}$ |
| | Test (small) | $8^*$ | $\{3, \ldots, 10$ discs$\}$ |
| | Test (large) | 18 | $\{15, 20, \ldots, 100$ discs$\}$ |
| $n$-puzzle | Train | $127^\dagger$ | $10 \times \{2$ grid size$\}, 60 \times \{3, 4$ grid size$\}$ |
| | Val | 3 | $3 \times \{5$ grid size$\}$ |
| | Test (small) | 20 | $10 \times \{3, 4$ grid size$\}$ |
| | Test (large) | 50 | $10 \times \{5, 6, 7, 8, 9$ grid size$\}$ |
| Sokoban | Train | 300 | $50 \times \{5, 7$ grid size$\} \times \{2$ boxes$\} \times \{3, 4, 5$ walls$\}$ |
| | Val | 3 | $3 \times \{8$ grid size$\} \times \{2$ boxes$\} \times \{3$ walls$\}$ |
| | Test (small) | 30 | $5 \times \{5, 7$ grid size$\} \times \{2$ boxes$\} \times \{3, 4, 5$ walls$\}$ |
| | Test (large) | 90 | $3 \times \{8, 9, 10, 11, 12$ grid size$\} \times \{2, 3$ boxes$\} \times \{3, 4, 5$ walls$\}$ |
| Spanner | Train | 810 | $10 \times \bigcup_{n=1,\ldots,10} \{(s, n)$ (spanners, nuts) $\mid s = n, \ldots, 10\}$ |
| | Val | 3 | $3 \times \{(11, 11)$ (spanners, nuts)$\}$ |
| | Test (small) | 75 | $5 \times \bigcup_{n=2,4,\ldots,8,10} \{(s, n)$ (spanners, nuts) $\mid s = n, n+2, \ldots, 8, 10\}$ |
| | Test (large) | 90 | $5 \times \{(15, 15), (20, 20), \ldots, (100, 100)$ (spanners, nuts)$\}$ |
| VisitAll | Train | 160 | $20 \times \{3, \ldots, 10$ grid size$\}$ |
| | Val | 3 | $3 \times \{11$ grid size$\}$ |
| | Test (small) | 40 | $5 \times \{3, \ldots, 10$ grid size$\}$ |
| | Test (large) | 90 | $5 \times \{15, 20, \ldots, 100$ grid size$\}$ |
| VisitSome | Train | 160 | $20 \times \{3, \ldots, 10$ grid size$\}$; goals $= \frac{1}{10}$(grid size)$^2$ |
| | Val | 3 | $3 \times \{11$ grid size$\}$; goals $= \frac{1}{10}$(grid size)$^2$ |
| | Test (small) | 40 | $5 \times \{3, \ldots, 10$ grid size$\}$; goals $= \frac{1}{10}$(grid size)$^2$ |
| | Test (large) | 90 | $5 \times \{15, 20, \ldots, 100$ grid size$\}$; goals $= \frac{1}{10}$(grid size)$^2$ |

both seen and unseen size testing in the domain-dependent training case. For the domain-independent training case, for each testing domain, we use the same training dataset consisting of the dataset from Sec. 5.1 excluding all testing domains, i.e. {Sec. 5.1 domains} \ {Blocksworld, ..., VisitSome}. The training pipeline and hyperparameters are fixed and the same for all evaluation methodologies as described in Sec. 5.1. More specifically, training time depends on the learning rate scheduler with termination once learning rate becomes too small. However, in a majority of the experiments, training takes no longer than 10 minutes. We reiterate that no hyperparameter tuning is performed on the models or the optimiser.

Given that training a model is a stochastic process and generalisation performance may

vary significantly based on the seed of the training pipeline, we train each model 5 times and select the best one. If we are evaluating for the unseen/large size category, we use a validation set of problems to evaluate and choose the best model by whichever model solves the most validation problems when applied to GBFS search, similarly to [Ferber et al., 2022]. The validation of models is done on a cluster with a single Intel Xeon 3.2 GHz CPU core and no GPUs and a 600 second timeout and 8GB memory. The validation problems are again outlined in Tab. 7.3, noting that they have size unseen in the training set. We break ties with the average expanded number of nodes on solved validation problems, and on the weighted loss from Eq. 5.3 if no problems were solved for all models. For the seen/small size category, we simply take the model with the best weighted loss. Validation metrics from our experiments for unseen/large size categories are reported in Sec. C in the appendix.

### 7.3.3 Baselines

The baselines we compare against are blind search, and both eager and lazy GBFS with $h^{\text{gc}}, h^{\max}, h^{\text{add}}$ and $h^{\text{FF}}$. All of these heuristics are also implemented in PWL. They are all run with a 600 second timeout and 8GB memory on a cluster with a single Intel Xeon 3.2 GHz CPU core.

Note that the hardware used for the baselines is different from the hardware for evaluating GOOSE due to resource constraints. The same constraints mean that we are not able to compare against the state-of-the-art heuristic learner STRIPS-HGN which will be left as future work. Our theoretical work suggests that GOOSE can learn more accurate heuristic functions which can generalise better. We also note that there are several planners which incorporate learning but their evaluation criteria are not as extensive or robust. Without mentioning specific works, the learning for planning solvers that are focused on performance usually have suboptimal evaluation criteria and exhibit one or both of the following traits.

- Solvers are not implemented efficiently and thus, termination of solvers is not done with runtime but instead on the number of expanded nodes. This is not an informative method to evaluate planners given that this method of evaluation does not prevent us from solving for $h^*$ on every state directly.
- In the learning domain-dependent heuristics setting, the size of unseen problems are not significantly larger than the sizes of training problems, up to at most 2 times their size. It may be the case that models can generalise well only up to a certain size before beginning to exhibit poor performance relative to classical planners but this information cannot be extrapolated from having small test problems.

Our set of experiments do not exhibit such traits. The search component of GOOSE is implemented in C++ and we have constructed search algorithms which effectively utilise GPUs to reduce overhead of evaluating neural network heuristic functions. Our comparison against classical planners is over the same runtime and we report three metrics: runtime, number of node expansions and plan quality. Furthermore, we constructed test

Table 7.4: Qualitative summary of results on large/unseen problems. Model entries are of the form $\alpha \rightarrow \beta$ indicating that training was done on problems with up to $\alpha$ objects and a model was able to solve problems with up to $\beta$ objects in the given 10 minute timeout. Domain-independent trained model entries have $\alpha = 0$ since they have not seen the domain during training. - indicates no problems could be solved. † indicates problems could be solved but performance is worse than classical heuristics.

| | Domain-dependent | | Domain-independent | |
|---|---|---|---|---|
| **Domain** | Ground | Lifted | Ground | Lifted |
| BLOCKSWORLD | 10→50 blocks | 10→35 blocks | 0→35 blocks | - |
| FERRY | 10→60 ferries | - | 0→45 ferries | - |
| GRIPPER | 10→100 balls | 10→85 balls | 0→75 balls | 0→65 balls |
| HANOI | - | - | - | - |
| $n$-PUZZLE | 4→6 grid size | - | 0→5 grid size | - |
| SOKOBAN | † | † | † | † |
| SPANNER | 10→25 spanners | 10→100 spanners | - | - |
| VISITALL | 10→50 grid size | 10→45 grid size | 0→55 grid size | 0→45 grid size |
| VISITSOME | 10→35 grid size | 10→15 grid size | 0→25 grid size | 0→25 grid size |

problems that are up to 10× the size of the training problems in order to find the limits in generalisation performance. We note that increasing the problem size exponentially increases the size of the search space. Hence the axes in plots corresponding to the number of expanded nodes of a solver are in log scale.

## 7.4 Results

We present and interpret results per domain with reference to each of the experimental configurations. The final section of the chapter comments on runtime using CPU and GPU hardware. We recall the dimensions of experimental configurations are

- the graph representations (SDG, $\text{SDG}^{\text{E}}$, FDG, $\text{FDG}^{\text{E}}$, LDG, $\text{LDG}^{\text{E}}$),
- the choice of GNN readout (sum or mean),
- the search algorithm (lazy or eager GBFS),
- the training setting (domain-dependent or domain-independent), and
- the evaluation setting (problems with seen/small or unseen/large sizes).

We refer to a GOOSE configuration, GOOSE model or just model as our GOOSE framework with a choice of graph representation, readout and search algorithm. Tab. 7.4 provides a condensed summary of generalisation capabilities of GOOSE on large/unseen size problems. Coverage tables of all the experiments, alongside cumulative coverage plots of plan quality and runtime per domain are also provided in Sec. C in the appendix.

Table 7.5: Coverage of solved instances on BLOCKSWORLD. The top 3 performing planners for each row are highlighted, with the best planner in bold.

| **Seen/small size** (40) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | 33 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 33 | 34 |
| | lazy | 33 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 37 | 31 | **40** | 32 | 23 |
| mean | eager | 33 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 33 | 34 |
| | lazy | 33 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 33 | 36 |
| **Unseen/large size** (90) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | - | 15 | - | 20 | 10 | 18 | **22** | 13 | 10 | 9 | 6 | 16 | - | 4 | 6 | - | - |
| | lazy | - | 13 | - | 21 | 10 | 17 | **23** | 14 | 10 | 8 | 7 | 15 | - | 7 | 8 | - | - |
| mean | eager | - | 15 | - | 20 | 10 | 25 | 26 | **32** | 31 | 19 | 8 | 6 | 4 | 9 | 10 | - | - |
| | lazy | - | 13 | - | 21 | 10 | 26 | 32 | **33** | 32 | 20 | 7 | 5 | 4 | 10 | 9 | - | - |

### 7.4.1 Blocksworld

In the large problem setting, we train on instances with up to 10 blocks and our test set consists of problems ranging from 15 to 100 blocks. Fig. 7.7 and 7.8 illustrate the cumulative coverage of solvers over number of expansions for problems of seen/small size and unseen/large size respectively. Tab. 7.5 provides the coverage table. We refer to Sec. C.5.1 in the appendix for additional cumulative coverage plots of solvers over runtime and plan cost.

We note that GOOSE models with grounded graphs and a mean readout solve significantly more difficult problems of up to 50 blocks and expand at least 1 and up to 2 orders of magnitude fewer nodes than classical heuristics. GOOSE is able to solve medium sized problems in a few seconds while classical planners require several minutes. GOOSE also return plans with better quality, with costs of plans half of those returned by classical heuristics.

Performance begins to drop at around 50 blocks where coverage is not as consistent. This suggests that the learners are not able to generalise effectively beyond that point, and may be limited by the receptive field (number of message passing layers) similarly to ASNets. This almost matches the performance of ASNets which is able to solve Blocksworld instances with 50 blocks consistently with training on instances with up to at most 10 blocks. It is worth performing a hyperparameter search and additional optimisations discussed later in Sec. 9.3 to see whether it is possible to match or go beyond this performance of ASNets. We also note that ASNets has access to predicate and schema information and uses different neural network weights for different schema. This is the case for our lifted graph representations, LDG and $\text{LDG}^{\text{E}}$ but not our grounded ones. Thus, it may also be possible to encode predicate information into graphs similarly to LDG and $\text{LDG}^{\text{E}}$ which does not interfere with domain-independent training. Lastly, ASNets leverages landmarks in its computation while GOOSE does not.

We note that our sum readout models are still more informative than classical heuristics as the best performing graph representation expands at least an order of magnitude fewer nodes than classical heuristics. However, the instability of training the sum readout make them less robust to larger problems as seen in their smaller coverage with respect to mean readout models. We note that the best performing mean readout GOOSE configuration solves `blocks50-task04` with a plan cost of 152 with 1982 expansions, but computes a heuristic of 50 at the initial state. In contrast, the best sum readout GOOSE model provides a heuristic value of 178 for the same problem but does not solve it. This suggests having higher heuristic estimates is not necessarily a good measure of heuristic quality for GBFS, given that GBFS uses a heuristic to rank states.

Furthermore, we see that lifted models perform worse than their grounded counterparts which suggest that their more compact representation removes some information when used with MPNNs which limits their expressivity on learning useful heuristics for Blocksworld.

When we consider domain-independent training, only the grounded graphs are able to solve the unseen problems. However, the learned heuristics are still informative as they perform better than blind search or heuristic search with $h^{\max}$ which are not able to solve any hard problems and in some cases match more informed classical heuristics $h^{\text{gc}}$ and $h^{\text{FF}}$.

We observe similar trends happening with the seen/small instances. Domain-dependent trained heuristics with either mean or sum readout are more informative and expand significantly fewer nodes than classical heuristics for almost all instances. Domain-independent trained heuristics on grounded graphs have around the same informativeness as classical heuristics with a mean readout. The informativeness is worse if we use lifted graphs or a sum readout instead.

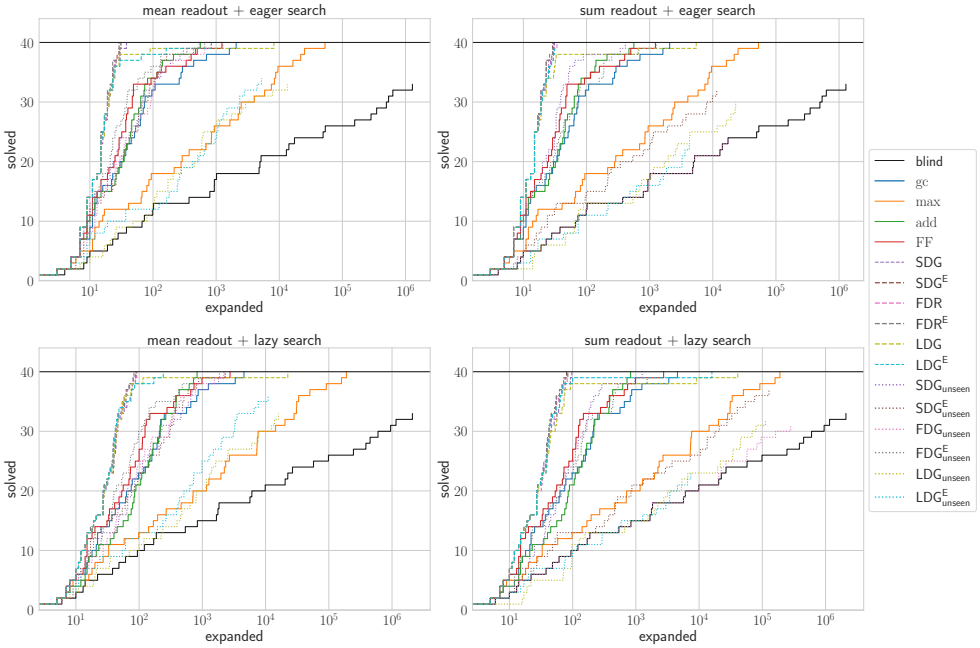



Figure 7.7: Cumulative coverage over number of expanded states on seen/small size Blocksworld instances. Total number of problems: 40.

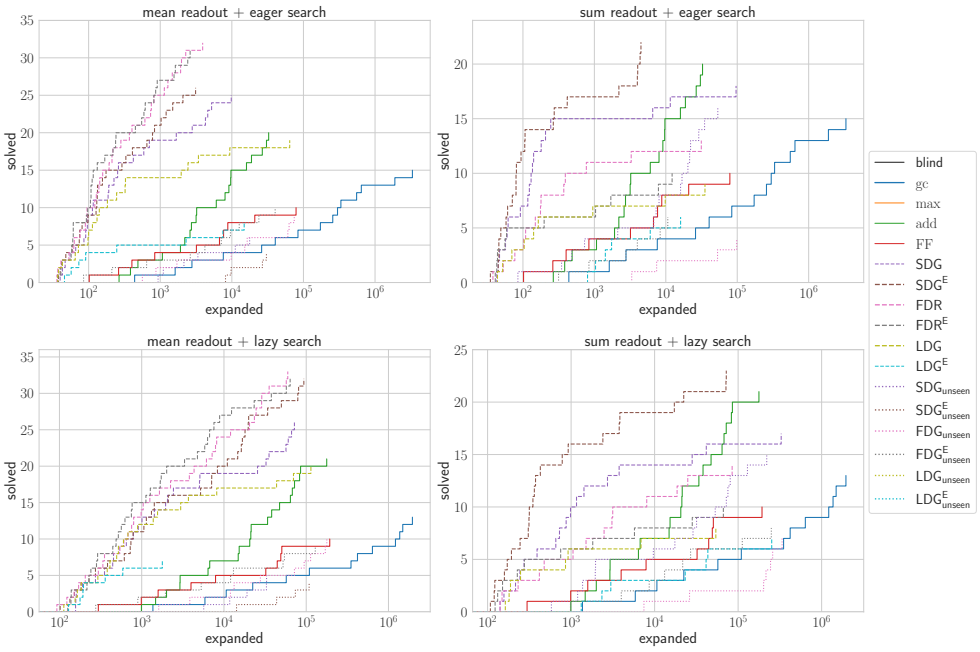



Figure 7.8: Cumulative coverage over number of expanded states on unseen/large size Blocksworld instances. Total number of problems: 90.

Table 7.6: Coverage of solved instances on FERRY. The top 3 performing planners for each row are highlighted, with the best planner in bold.

| **Seen/small size** (125) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | 90 | **125** | 79 | **125** | **125** | **125** | **125** | **125** | **125** | 112 | 65 | **125** | 115 | 40 | **125** | 56 | 40 |
| | lazy | 88 | **125** | 82 | **125** | **125** | **125** | **125** | **125** | **125** | 118 | 82 | **125** | 119 | 59 | **125** | 59 | 41 |
| mean | eager | 90 | **125** | 79 | **125** | **125** | **125** | **125** | **125** | **125** | 116 | 107 | **125** | **125** | 113 | **125** | 97 | 78 |
| | lazy | 88 | **125** | 82 | **125** | **125** | **125** | **125** | **125** | **125** | 118 | 117 | **125** | **125** | 109 | **125** | 96 | 79 |
| **Unseen/large size** (90) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | - | **90** | - | 11 | 38 | 9 | 43 | 20 | 41 | - | - | 44 | - | 1 | 17 | - | - |
| | lazy | - | **90** | - | 16 | 69 | 11 | 40 | 19 | 40 | - | - | 44 | - | - | 13 | - | - |
| mean | eager | - | **90** | - | 11 | 38 | 35 | 40 | 39 | 40 | - | - | 14 | 5 | 30 | 12 | - | - |
| | lazy | - | **90** | - | 16 | 69 | 33 | 40 | 40 | 40 | - | - | 13 | 5 | 28 | 12 | - | - |

### 7.4.2 Ferry

In the large problem setting, we train on instances with up to 10 locations and cars, and test on problems with between 15 to 100 locations and cars. Fig. 7.9 and 7.10 illustrate the cumulative coverage of solvers over number of expansions for problems of seen/small size and unseen/large size respectively. Tab. 7.6 provides the coverage table. We refer to Sec. C.5.2 in the appendix for additional cumulative coverage plots of solvers over runtime and plan cost.

For Ferry, the goal count heuristic $h^{\text{gc}}$ has the best coverage due to its fast evaluation. It quickly and greedily searches the fastest way to get any remaining car not at its target location to its goal which is a good strategy for this domain. It is able to solve any instance in under 30 seconds with eager search and provides the best quality plans.

In terms of informativeness, GOOSE with edge labelled grounded graphs and sum readouts provides the most informative heuristics with the fewest expanded number of nodes on problems of up to 60 ferries. We note that beyond 60 problems, we run into problems where batched graphs do not fit into the memory of the GPU and the solver terminates. This may be fixed in future work by detecting when the graph data becomes too large to fit into the GPU and splitting the batches for evaluation. Runtime wise, GOOSE is usually twice as fast as $h^{\text{FF}}$ with eager search but outperformed with lazy search where lazy search does not benefit GOOSE for this domain. Furthermore, GOOSE returns similar cost plans to $h^{\text{gc}}$ and $h^{\text{FF}}$ with eager search, but better plans than $h^{\text{FF}}$ with lazy search.

Contrary to Blocksworld, sum readout models perform better than mean readout models. For example, GOOSE with a sum readout, $\text{SDG}^{\text{E}}$ and eager search provides a heuristic estimate of 189 for the instance `ferry-l50-c50` and solves it with plan cost 190 and 894 expansions, whereas its mean readout counterpart also similarly provides a plan cost of 186, but provides a heuristic estimate of 186 and requires 5400 expansions. Thus, it

is desirable to have higher variance in the distribution of heuristic estimates when the ranking is accurate.

We also note that some domain-independent trained heuristics have reasonable heuristic estimates with informativeness greater than $h^{\max}$ or blind search. With regards to seen/small instances, domain-dependent trained heuristics are marginally the most informative heuristics on all instances. Lifted graphs underperform as they cannot learn to solve both small or large instances.

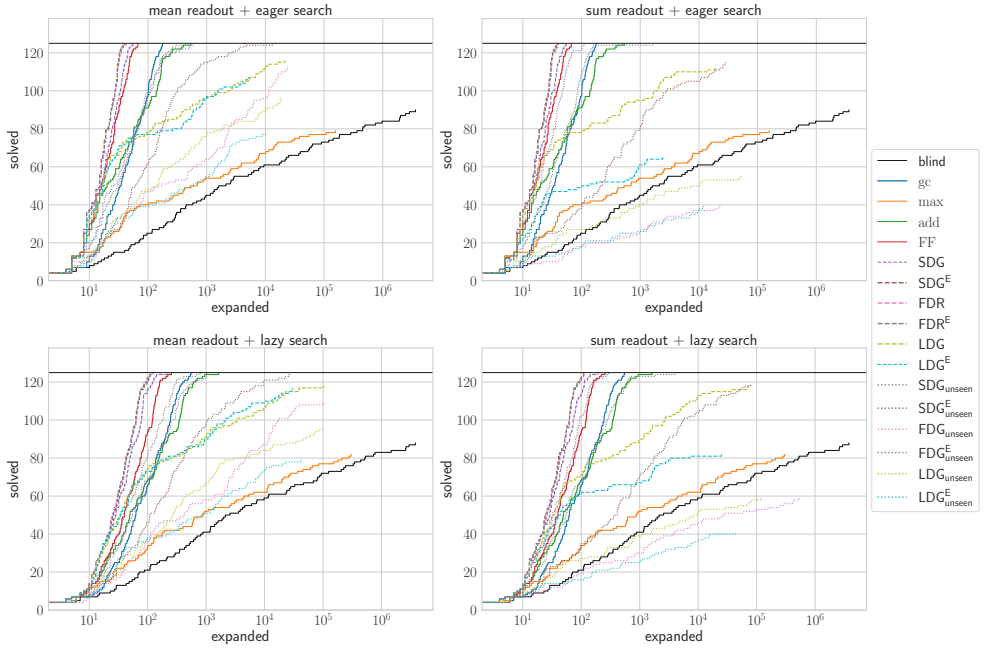



Figure 7.9: Cumulative coverage over number of expanded states on seen/small size Ferry instances. Total number of problems: 125.

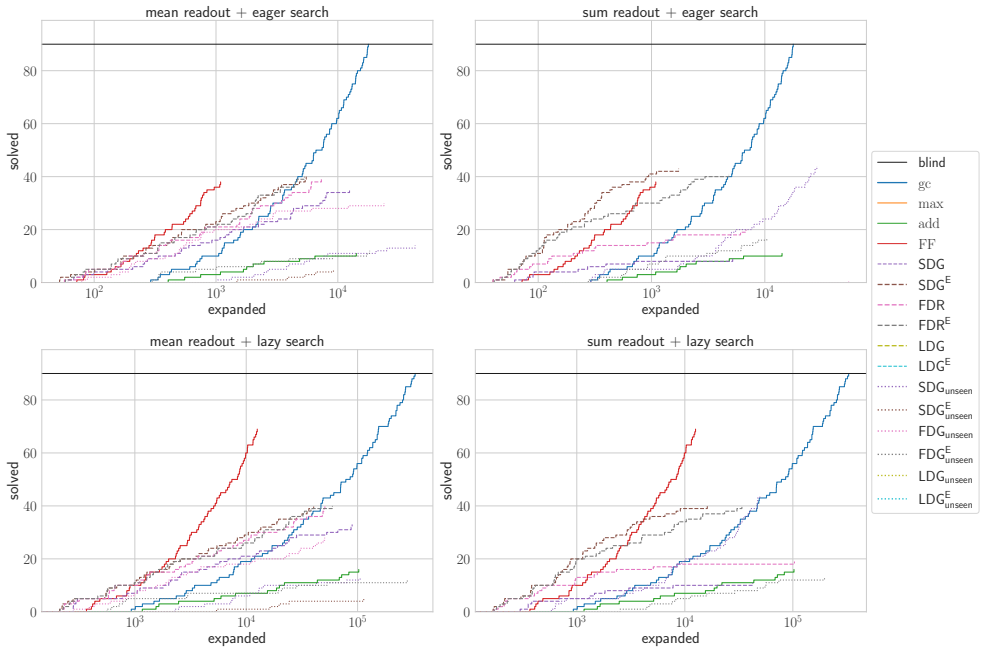



Figure 7.10: Cumulative coverage over number of expanded states on unseen/large size Ferry instances. Total number of problems: 90.

Table 7.7: Coverage of solved instances on GRIPPER. The top 3 performing planners for each row are highlighted, with the best planner in bold.

| **Seen/small size** (10) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | 9 | **10** | **10** | **10** |
| | lazy | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** |
| mean | eager | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | 8 | **10** | **10** | 9 | **10** |
| | lazy | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | 9 | **10** | **10** | **10** | **10** |
| **Unseen/large size** (18) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | 1 | **18** | - | **18** | 14 | 7 | 1 | 5 | 3 | 14 | 12 | 4 | - | - | - | 3 | 7 |
| | lazy | 1 | **18** | - | **18** | **18** | 10 | 12 | 17 | **18** | 15 | 12 | 5 | 1 | - | 13 | 7 | 9 |
| mean | eager | 1 | **18** | - | **18** | 14 | 7 | 7 | 5 | 7 | 7 | 6 | 2 | - | 4 | 2 | 3 | 5 |
| | lazy | 1 | **18** | - | **18** | **18** | 7 | 8 | 5 | 7 | 7 | 6 | 2 | - | 4 | 1 | 6 | 8 |

### 7.4.3 Gripper

In the large problem setting, we train on instances with up to 10 balls and cars, and test on problems with between 15 to 100 balls. Fig. 7.11 and 7.12 illustrate the cumulative coverage of solvers over number of expansions for problems of seen/small size and unseen/large size respectively. Tab. 7.7 provides the coverage table. We refer to Sec. C.5.3 in the appendix for additional cumulative coverage plots of solvers over runtime and plan cost.

For Gripper, the $h^{\text{add}}$ heuristic has the best coverage due to its almost perfect heuristic estimates. This is because $h^{\text{add}}$ assumes each goal is achieved independently from each other and this is almost the case with Gripper. GOOSE models trained on small instances have very accurate $h^*$ estimates but expand only marginally fewer nodes than $h^{\text{add}}$ on the same small instances.

We note that the GOOSE with lifted graphs and sum readout generalise the best with eager search as they have the highest coverage and lowest number of expansions among all GOOSE configurations. On the other hand, the grounded graphs perform worse which may suggest that they overfit on the training set and were not learning the semantics of the problem. We also note that domain-independent trained $\text{LDG}^{\text{E}}$ with mean readout outperforms its domain-dependent trained grounded counterparts. This may also suggest that knowing the predicates and objects of the problem may be useful for the grounded graph representations.

We also see that lazy search significantly improves the performance of sum readout models, allowing $\text{FDG}^{\text{E}}$ graphs to reach full coverage when its eager counterpart only solves 3 problems. This may be due the regularising nature of lazy search which makes heuristics less informed and in turn dampens the effects of incorrect heuristic estimates. Nevertheless, classical heuristics tend to have significantly lower runtimes than GOOSE in both eager and lazy search. Goal count can solve any gripper problem in a few seconds,

while the best performing GOOSE model takes up to 250 seconds for the most difficult problem.

Mean readout models appear to perform a blind search until a certain number of balls have reached the other room, in which case the problems become similar to trained instances where the model then rapidly reaches the goal.

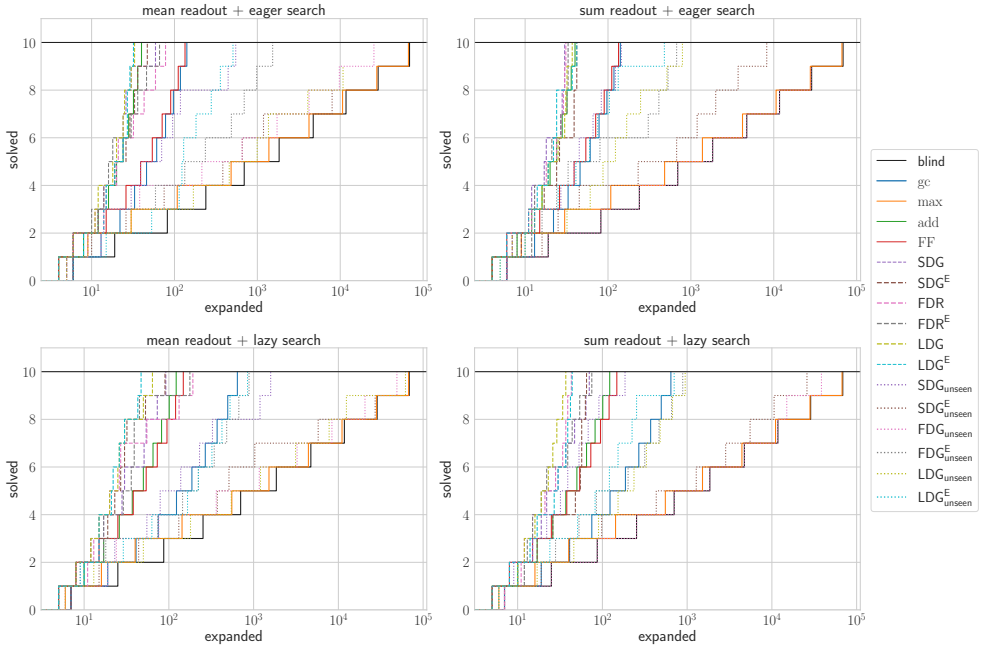



Figure 7.11: Cumulative coverage over number of expanded states on seen/small size Gripper instances. Total number of problems: 10.

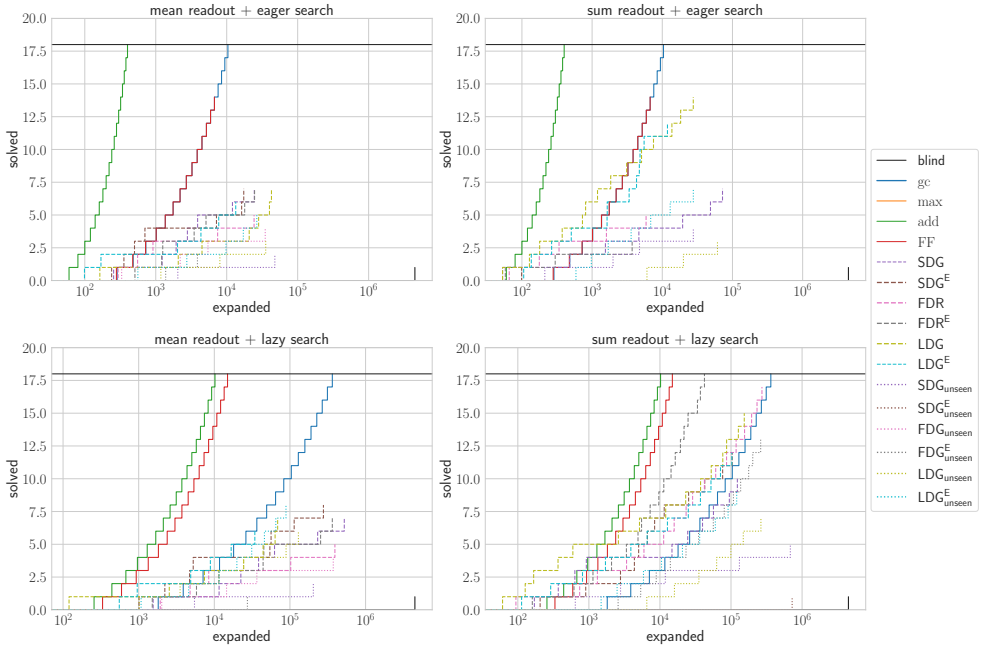



Figure 7.12: Cumulative coverage over number of expanded states on unseen/large size Gripper instances. Total number of problems: 18.

Table 7.8: Coverage of solved instances on HANOI. The top 3 performing planners for each row are highlighted, with the best planner in bold.

| **Seen/small size** (8) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^E}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^E}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^E}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^E}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^E}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^E}$ |
| sum | eager | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | 6 |
| | lazy | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | 7 |
| mean | eager | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | 7 | 7 | **8** | 6 | **8** | 7 | 7 | 6 |
| | lazy | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | 7 | **8** | 7 | **8** | 7 | **8** | **8** | **8** | 6 |
| **Unseen/large size** (18) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^E}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^E}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^E}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^E}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^E}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^E}$ |
| sum | eager | **1** | **1** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| | lazy | **1** | **1** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| mean | eager | **1** | **1** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| | lazy | **1** | **1** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |

### 7.4.4 Hanoi

In the large problem setting, we train on instances with up to 10 discs, and test on problems with between 15 to 100 discs. Fig. 7.13 illustrates the cumulative coverage of solvers over number of expansions for problems of seen/small size respectively. The plot for unseen/large size problems is omitted due to only one instance being solved by two planners. Tab. 7.8 provides the coverage table. We refer to Sec. C.5.4 in the appendix for additional cumulative coverage plots of solvers over runtime and plan cost.

Unfortunately, the large problems we chose were too difficult for any planner to solve except for blind search and search with goal count. This is because the optimal plan length is $2^d - 1$ where $d$ is the number of discs of the problem which suggests that heuristics must be very well informed if they have a significant computational cost to be able to find such exponentially long plans. It may be useful for future work to construct Hanoi instances with initial states closer to the goal state.

When we observe the training loss of our GOOSE models in Sec. C.1, we see that it struggles to learn even on the training set whose ground truth value range up to 1023. This can be attributed to the small receptive field of MPNNs relative to the search space of Hanoi problems and the difficulty of learning recursively defined plans.

However, we can observe some interesting results for smaller Hanoi instances. Some configurations of domain-independent trained heuristics outperform the baseline heuristics in terms of number of expanded nodes. For instance, we see that $h^{\text{gc}}$ is the most informative heuristic out of all the classical heuristics we consider. However, it is not by more than an order of magnitude better for any of the problems. Furthermore, $h^{\text{add}}$ expands more nodes than blind search on the Hanoi instance with 10 discs. Lastly despite expanding many more nodes than GOOSE, blind search and $h^{\text{gc}}$ are able to solve all small Hanoi instances in less than 3 seconds each while the best performing GOOSE model takes up to 15 seconds.

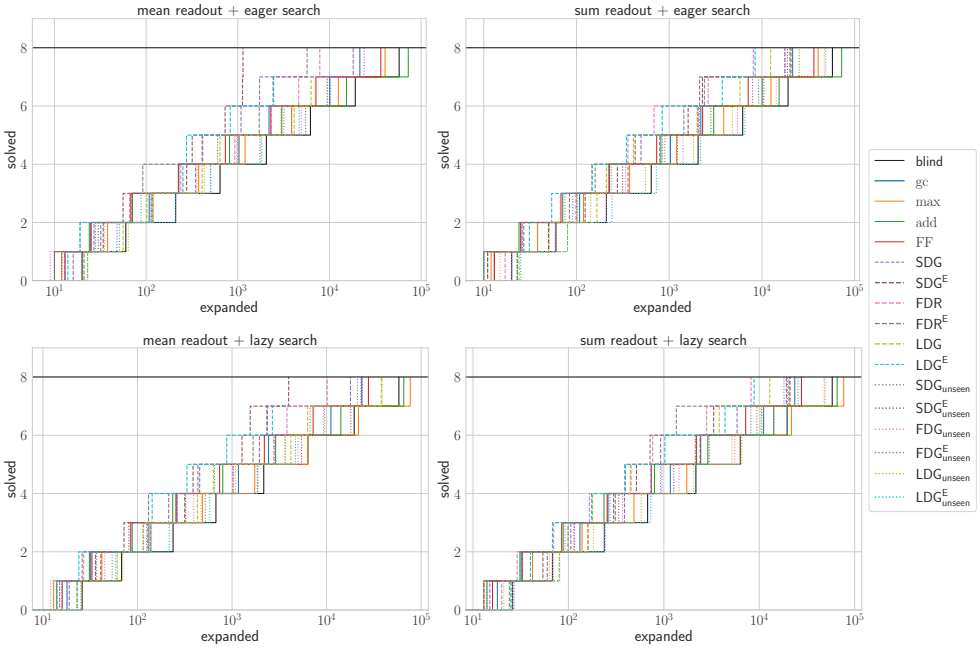



Figure 7.13: Cumulative coverage over number of expanded states on seen/small size HANOI instances. Total number of problems: 8.

Table 7.9: Coverage of solved instances on $n$-PUZZLE. The top 3 performing planners for each row are highlighted, with the best planner in bold.

| **Seen/small size** (20) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | 10 | **20** | 10 | 19 | **20** | **20** | **20** | **20** | **20** | 5 | 1 | **20** | 17 | 12 | **20** | 4 | 1 |
| | lazy | 10 | **20** | 10 | 19 | **20** | **20** | **20** | **20** | **20** | 10 | 8 | **20** | **20** | 11 | **20** | 10 | 5 |
| mean | eager | 10 | **20** | 10 | 19 | **20** | **20** | **20** | **20** | **20** | 2 | 1 | 19 | 14 | **20** | **20** | 3 | 1 |
| | lazy | 10 | **20** | 10 | 19 | **20** | **20** | **20** | **20** | **20** | 10 | 5 | **20** | 19 | **20** | **20** | 8 | 3 |
| **Unseen/large size** (50) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | - | **11** | - | - | **11** | 10 | 8 | - | 10 | - | - | 6 | - | - | 5 | - | - |
| | lazy | - | 11 | - | - | **14** | 12 | 12 | 5 | 13 | - | - | 7 | - | - | 6 | - | - |
| mean | eager | - | **11** | - | - | **11** | 5 | 8 | 8 | 6 | - | - | 6 | - | 8 | 5 | - | - |
| | lazy | - | 11 | - | - | **14** | 8 | 8 | 9 | 9 | - | - | 3 | - | 8 | 7 | - | - |

### 7.4.5 $n$-puzzle

In the large problem setting, we train on instances with grid size up to 4, and test on problems with between 5 and 9 grid size inclusive. Fig. 7.14 and 7.15 illustrate the cumulative coverage of solvers over number of expansions for problems of seen/small size and unseen/large size respectively. Tab. 7.9 provides the coverage table. We refer to Sec. C.5.5 in the appendix for additional cumulative coverage plots of solvers over runtime and plan cost.

We note that $h^{\text{FF}}$ and $h^{\text{gc}}$ are the best performing heuristics, although $h^{\text{FF}}$ expands around 2 orders of magnitude fewer nodes than $h^{\text{gc}}$. They are able to solve all puzzles with grid size 5, and some puzzles with grid size 6. The case is similar for GOOSE models with lazy search and sum readout. They are not as informative as $h^{\text{FF}}$ as they expand more nodes but for almost all problems that both solve, the plan quality of GOOSE models are better. We also note that some of domain-independent trained models with SDG, FDG and FDG$^{\text{E}}$ provide useful heuristic estimates which allow them to solve some hard problems, in contrary to $h^{\max}$ and $h^{\text{add}}$ which do not solve any. Mean readout models are less informative than their sum readout counterparts but still return better quality plans. Unfortunately, lifted models perform poorly as they are unable to learn from training data shown by their high training loss, see Sec. C.1.

On seen/small size test instances, $h^{\text{FF}}$ and grounded graph GOOSE models have similar informedness and node expansions. However, learned heuristics still provide better quality plans with cost around 75% that of $h^{\text{FF}}$ plans. Lifted graph models have informedness on the same level as blind search regardless of domain-dependent or independent training. Thus, they do not provide any meaningful guidance.

With regards to runtime, $h^{\text{gc}}$ and $h^{\text{FF}}$ are almost always faster than GOOSE for both seen/small and unseen/large size problems. This is due to $h^{\text{gc}}$'s fast heuristic evaluations and $h^{\text{FF}}$ more informed heuristics.

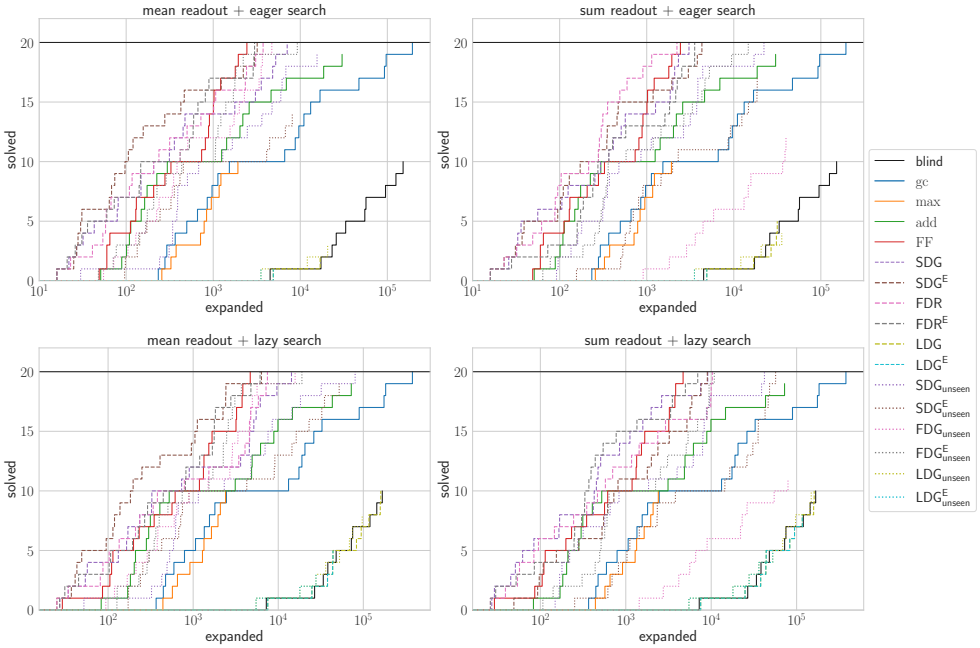



Figure 7.14: Cumulative coverage over number of expanded states on seen/small size $n$-PUZZLE instances. Total number of problems: 20.

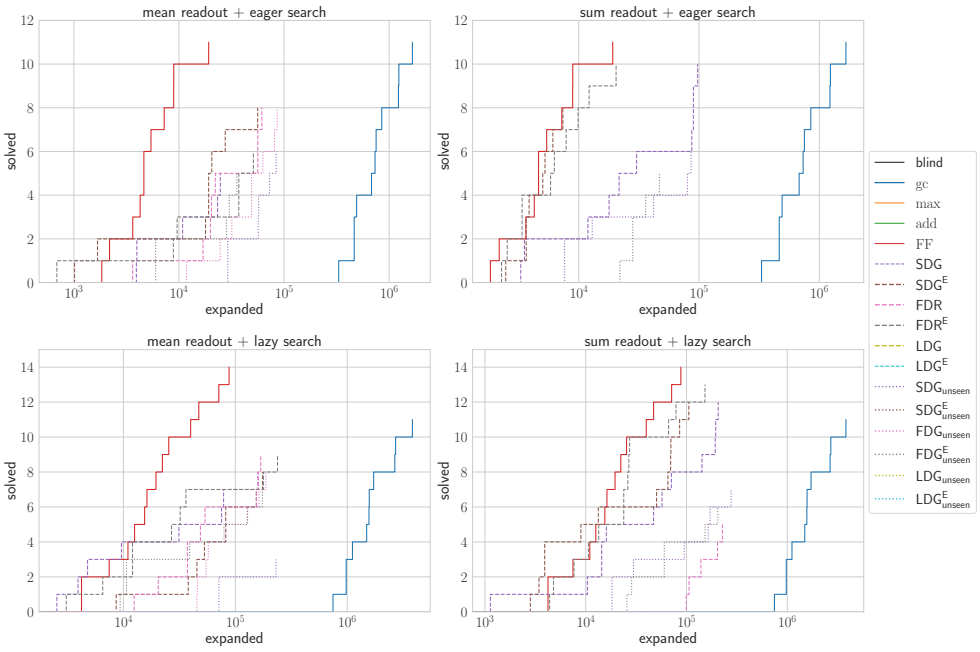



Figure 7.15: Cumulative coverage over number of expanded states on unseen/large size $n$-PUZZLE instances. Total number of problems: 50.

Table 7.10: Coverage of solved instances on SOKOBAN. The top 3 performing planners for each row are highlighted, with the best planner in bold.

| **Seen/small size** (30) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Readout** | **Search** | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| sum | eager | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | 18 | 23 | 23 | **30** | **30** |
| | lazy | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** |
| mean | eager | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | 27 | **30** | 17 | 15 | 23 | 15 | 27 |
| | lazy | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** |
| **Unseen/large size** (90) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
| **Readout** | **Search** | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| sum | eager | 42 | 81 | **82** | 52 | 45 | 30 | 31 | 29 | 34 | 27 | 27 | 20 | 12 | 3 | 18 | 18 | 18 |
| | lazy | 42 | 81 | **90** | 63 | 72 | 32 | 34 | 29 | 34 | 33 | 27 | 39 | 18 | 18 | 18 | 25 | 24 |
| mean | eager | 42 | 81 | **82** | 52 | 45 | 38 | 31 | 39 | 38 | 28 | 27 | 9 | 4 | 30 | 3 | 36 | 36 |
| | lazy | 42 | 81 | **90** | 63 | 72 | 35 | 39 | 45 | 35 | 31 | 31 | 20 | 18 | 37 | 18 | 36 | 36 |

### 7.4.6 Sokoban

In the large problem setting, we train on instances with 2 boxes and grid size up to 7, and test on problems with between 8 to 12 grid size and 2 or 3 boxes. Fig. 7.16 and 7.17 illustrate the cumulative coverage of solvers over number of expansions for problems of seen/small size and unseen/large size respectively. Tab. 7.10 provides the coverage table. We refer to Sec. C.5.6 in the appendix for additional cumulative coverage plots of solvers over runtime and plan cost.

We note that $h^{\max}$ has the overall best performance. This may be attributed to its conservative estimates by only solving for the most difficult Sokoban goal which in turn does not guide the search algorithm into incorrect search regions in the complex Sokoban domain. This is in contrast to $h^{\text{add}}$ and $h^{\text{FF}}$ which are greedier delete relaxation heuristics. These heuristics attempt to move all boxes to their goal location with no regard to the interactions between boxes which in turn may provide poorly informed estimates and guide the planner to unpromising search regions. Our learned heuristics provide better informativeness for smaller test problems but are unable to generalise to larger problems well where they expand significantly more nodes than delete relaxation heuristics.

For the set of unseen/large size problems, most GOOSE configurations perform worse than blind search in terms of coverage due to the slow heuristic evaluations. Lazy search slightly alleviates the limited batching performed in eager search as seen in the improvement in coverage for some problems. The fewer number of expansions when we using lazy search over eager search further suggests that GOOSE may provide poor heuristic estimates that guide the solver to unpromising states during search, given that lazy search can act as a regulariser. Furthermore, plan quality is not significantly better to classical heuristics, and becomes poorer with sum readout for larger problems.

Domain-independent trained heuristics provide informedness similar to goal count. One notable observation is that domain-independent trained heuristics on lifted graphs per-

form better than their domain-dependent trained counterparts when using mean readouts. Edge labelled grounded graphs provide the best informedness for seen size instances and expand at most an order of magnitude fewer nodes than $h^{\max}$ for most instances.

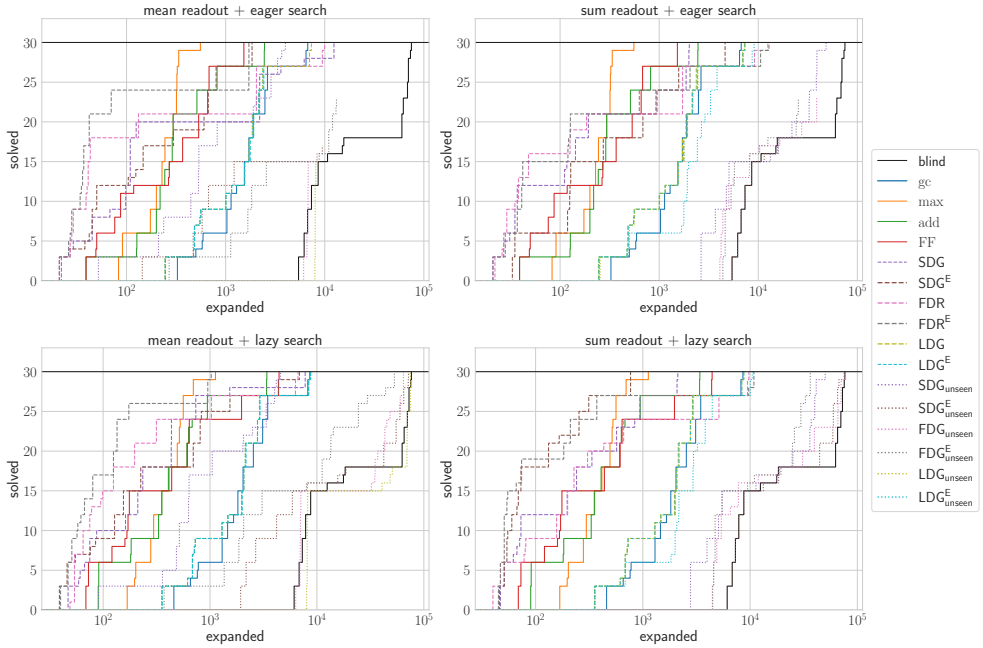



Figure 7.16: Cumulative coverage over number of expanded states on seen/small size Sokoban instances. Total number of problems: 30.

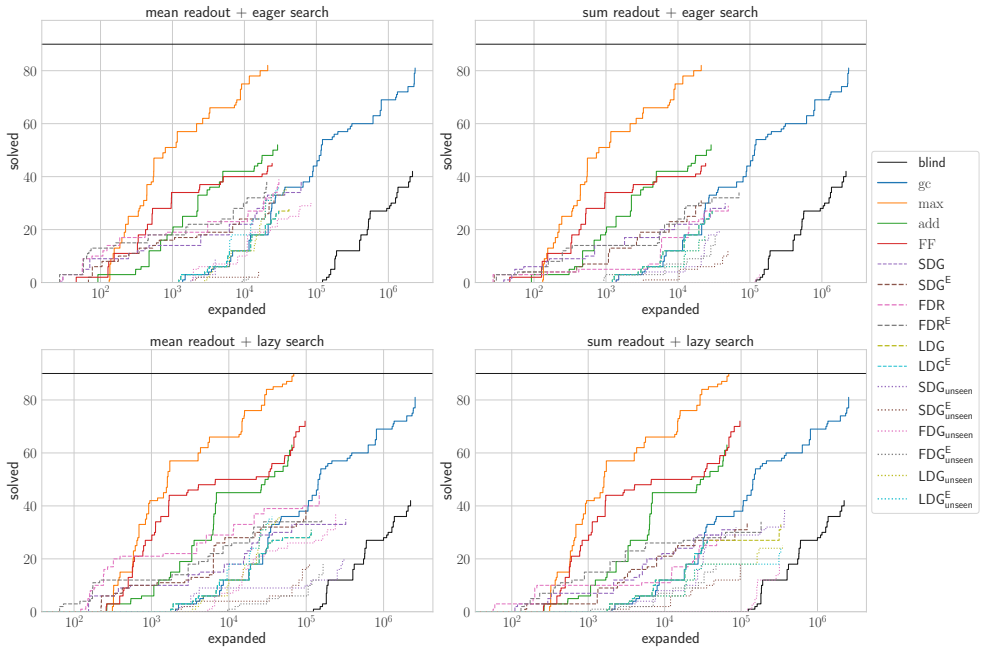



Figure 7.17: Cumulative coverage over number of expanded states on unseen/large size Sokoban instances. Total number of problems: 90.

Table 7.11: Coverage of solved instances on SPANNER. The top 3 performing planners for each row are highlighted, with the best planner in bold.

| **Seen/small size** (75) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | 65 | 65 | 60 | 60 | 60 | 70 | **75** | 72 | 70 | **75** | **75** | 60 | 40 | 40 | 60 | 55 | 40 |
| | lazy | 65 | 65 | 69 | 70 | 70 | **75** | **75** | 74 | **75** | **75** | **75** | 60 | 56 | 60 | 65 | 55 | 40 |
| mean | eager | 65 | 65 | 60 | 60 | 60 | 70 | 70 | 70 | 69 | **75** | **75** | 58 | 52 | 70 | 61 | 55 | 50 |
| | lazy | 65 | 65 | 69 | 70 | 70 | **75** | **75** | **75** | **75** | **75** | **75** | 60 | 58 | 70 | 70 | 55 | 50 |
| **Unseen/large size** (90) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | - | - | - | - | - | - | - | 1 | - | 15 | **50** | - | - | - | - | - | - |
| | lazy | - | - | - | - | - | 10 | 3 | 10 | 1 | 45 | **90** | - | - | - | - | - | - |
| mean | eager | - | - | - | - | - | - | - | - | - | **15** | 9 | - | - | - | - | - | - |
| | lazy | - | - | - | - | - | 1 | - | 2 | 1 | **25** | 9 | - | - | - | - | - | - |

### 7.4.7 Spanner

In the large problem setting, we train on instances with up to 10 spanners and nuts, and test on problems with between 15 to 100 objects. Fig. 7.18 and 7.19 illustrate the cumulative coverage of solvers over number of expansions for problems of seen/small size and unseen/large size respectively. Tab. 7.11 provides the coverage table. We refer to Sec. C.5.7 in the appendix for additional cumulative coverage plots of solvers over runtime and plan cost.

Classical heuristics and domain-independent trained heuristics do not solve any large Spanner problems. However lifted graphs have great performance with $\mathsf{LDG}^{\mathsf{E}}$ solving all instances with sum readout and lazy search. It is able to solve any instance in under 30 seconds. The reason why the same model with eager search does not solve all the instances is because at the end of the corridor, it evaluates up to $sn$ states at a time corresponding to all possible ways of using spanners to fix one nut. The number of evaluations becomes a bottleneck even for the perfect heuristic. Lazy search does not have this problem due to its deferred evaluation.

Similarly to Gripper, lifted graphs are able to generalise significantly better than their grounded counterparts. This could be explained by either overfitting of the grounded graph models as seen in their perfect training loss, or predicate and action schema information helping with generalisation of lifted graphs and missing in grounded graphs. The latter explanation may be more feasible given that it could be the case that the lifted models are learning to count spanners with the aid of the spanner predicate, whereas grounded models may not be able to learn which propositions correspond to spanners.

Grounded models generalise poorer even on seen and small size test instances if we observe the number of expansions with eager search, with one exception of GOOSE with $\mathsf{SDG}^{\mathsf{E}}$ and a sum readout. However, using lazy search reverses the story with grounded graphs expanding fewer nodes. Viewing lazy search as a regulariser further supports the

argument of grounded graphs overfitting. Lastly, we note that unlike the large instances, some small instances contain more spanners than nuts where the sum readout graphs may not learn to take only the required number of spanners instead of all the spanners.

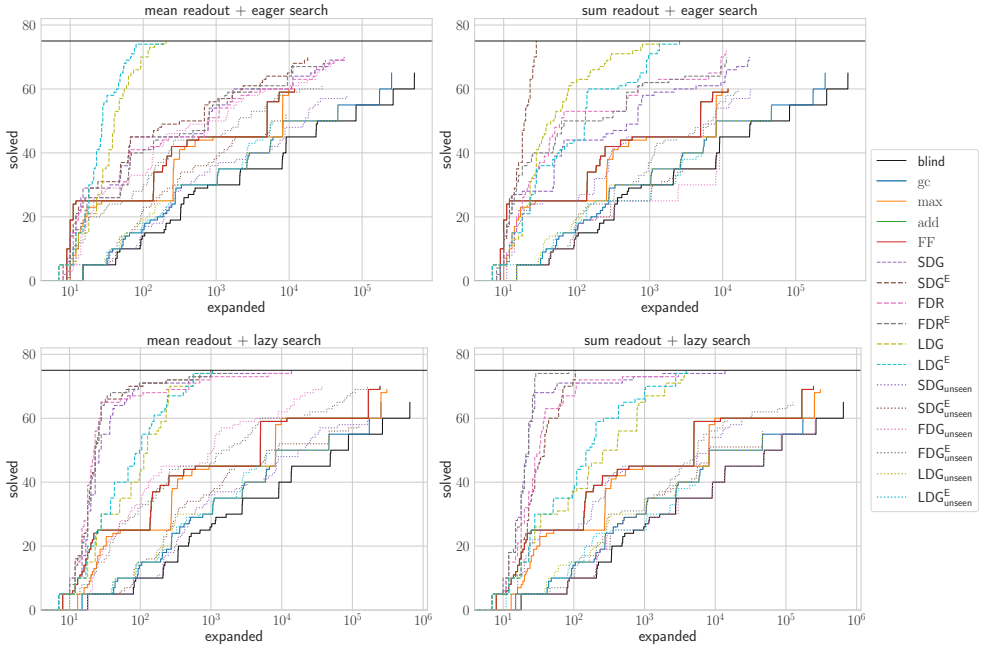



Figure 7.18: Cumulative coverage over number of expanded states on seen/small size Spanner instances. Total number of problems: 75.

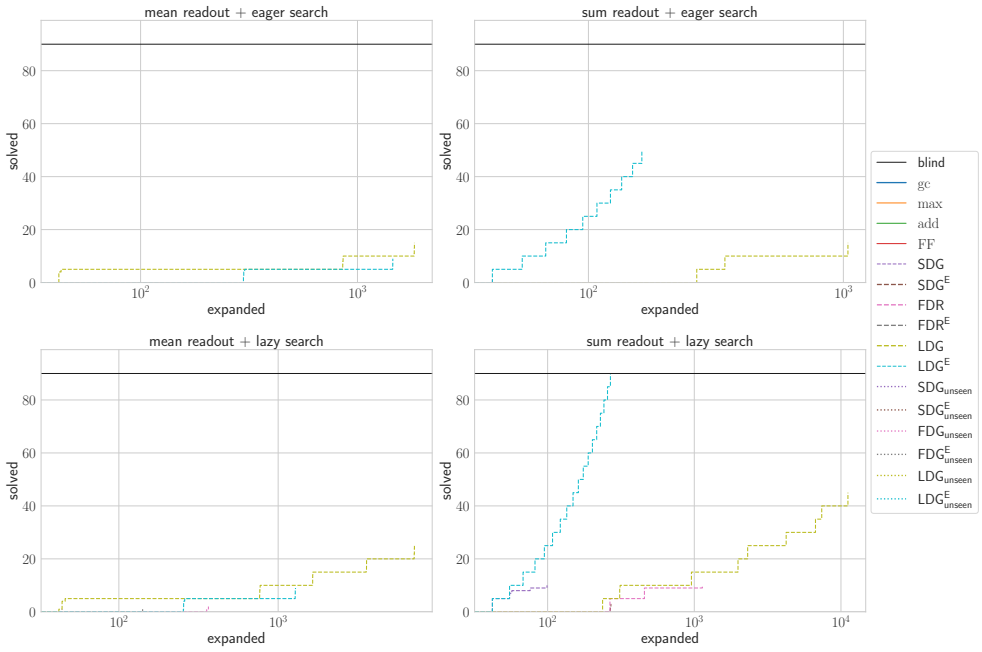



Figure 7.19: Cumulative coverage over number of expanded states on unseen/large size Spanner instances. Total number of problems: 90.

Table 7.12: Coverage of solved instances on VisitAll. The top 3 performing planners for each row are highlighted, with the best planner in bold.

| **Seen/small size** (40) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^E}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^E}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^E}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^E}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^E}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^E}$ |
| sum | eager | 10 | **40** | 10 | **40** | 39 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 10 | 13 | 31 | **40** | **40** |
| | lazy | 10 | **40** | 10 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 15 | 15 | 34 | **40** | **40** |
| mean | eager | 10 | **40** | 10 | **40** | 39 | **40** | **40** | **40** | **40** | **40** | **40** | 14 | 31 | 20 | 13 | **40** | **40** |
| | lazy | 10 | **40** | 10 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 16 | 36 | 20 | 15 | **40** | **40** |
| **Unseen/large size** (90) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^E}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^E}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^E}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^E}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^E}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^E}$ |
| sum | eager | - | **75** | - | 2 | 9 | 38 | 39 | 21 | 20 | 31 | 31 | 44 | 30 | - | 16 | 17 | 30 |
| | lazy | - | **73** | - | 4 | 14 | 40 | 30 | 24 | 17 | 30 | 25 | 40 | 28 | - | 17 | 15 | 25 |
| mean | eager | - | **75** | - | 2 | 9 | 15 | 15 | 10 | 5 | 10 | 10 | - | - | - | - | 12 | 12 |
| | lazy | - | **73** | - | 4 | 14 | 15 | 15 | 10 | 5 | 10 | 10 | - | - | - | - | 15 | 15 |

### 7.4.8 VisitAll

In the large problem setting, we train on instances with grid size up to 10, and test on problems with grid size between 15 and 100. Fig. 7.20 and 7.21 illustrate the cumulative coverage of solvers over number of expansions for problems of seen/small size and unseen/large size respectively. Tab. 7.12 provides the coverage table. We refer to Sec. C.5.8 in the appendix for additional cumulative coverage plots of solvers over runtime and plan cost.

VisitAll is another problem which the goal count heuristic excels at since it is close to the perfect heuristic for the domain and is fast to compute. We observe that our sum readout models generalise well, up to problems with grid size 50, given that it is easy to learn the goal count heuristic as an approximation for $h^*$. All domain-dependent trained GOOSE models have coverage greater than the remaining classical heuristics such as $h^{\text{FF}}$ and $h^{\text{add}}$.

For this domain, the bottleneck of GOOSE is the time to evaluate heuristics, given that the models are generalising well for large problems when observing the cumulative coverage over expanded nodes with expansions similar to $h^{\text{gc}}$. We also note that the GOOSE's plan quality also matches that of $h^{\text{gc}}$. However, we reach the timeout at smaller instances than $h^{\text{gc}}$.

It is more difficult for mean readout models to generalise given that they are limited by information loss through averaging features during readout. Nevertheless, they generalise well to problems of the same size as the train set with high informedness. We further note that the best GOOSE models are the domain-independent trained models with marginally better coverage and fewer expansions than $h^{\text{gc}}$ with eager search. This may suggest that GOOSE trained over a wide variety of domains in the domain-independent setting learns a modification of goal count which is helpful for VisitAll.

Interestingly, FDR graph representations perform worst out of all GOOSE models com-

pared to STRIPS and lifted representations. This could be attributed to overfitting to additional computed FDR structures not present in STRIPS and the lifted graphs. We also see that FDR models have lower training loss than other models even after the validation criteria.

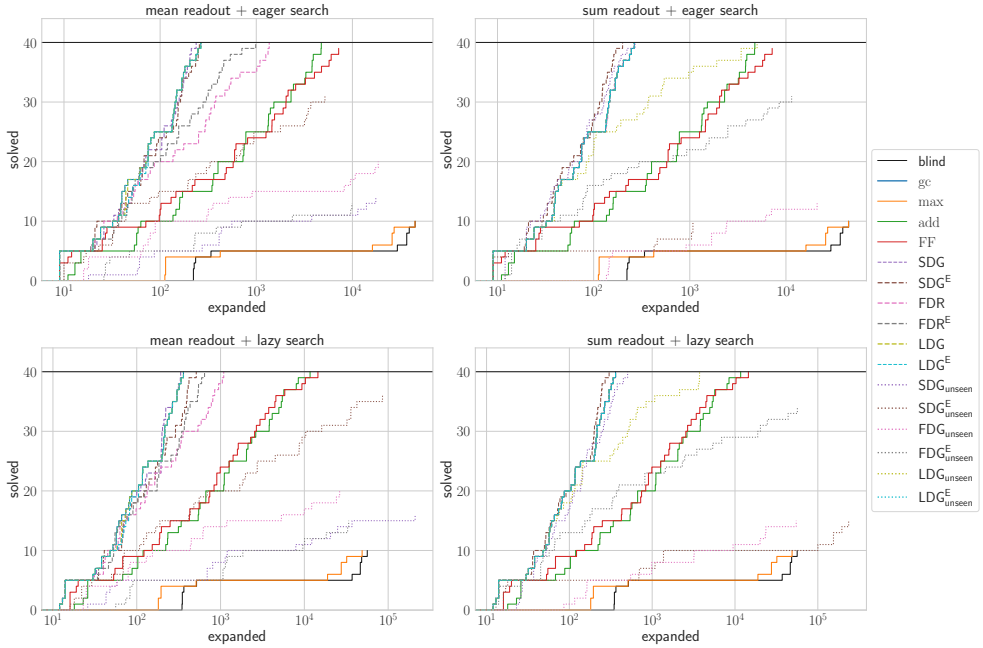



Figure 7.20: Cumulative coverage over number of expanded states on seen/small size VisitAll instances. Total number of problems: 40.

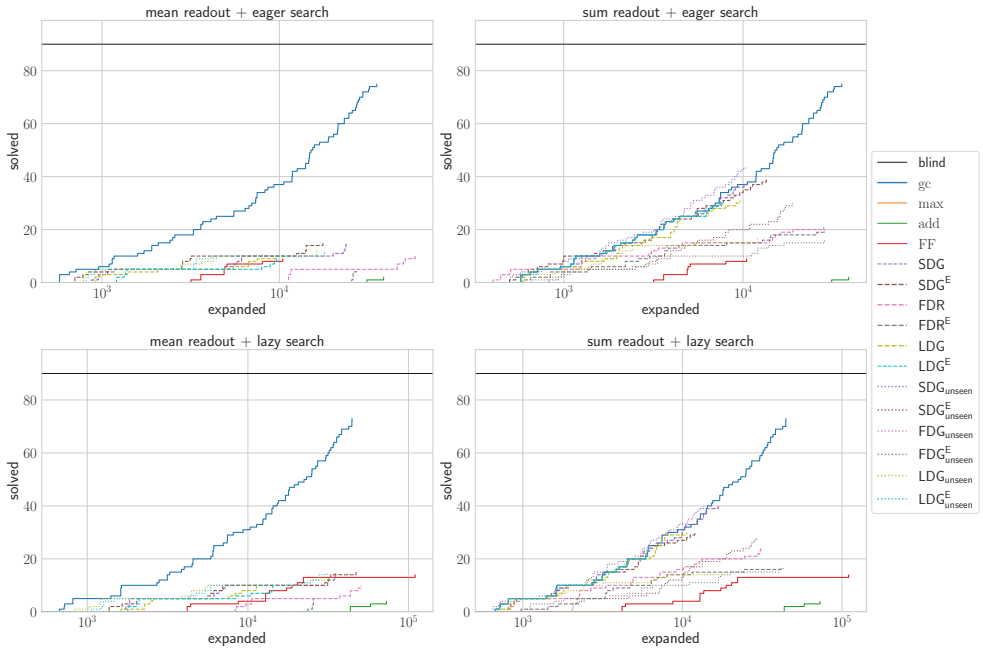



Figure 7.21: Cumulative coverage over number of expanded states on unseen/large size VisitAll instances. Total number of problems: 90.

Table 7.13: Coverage of solved instances on VisitSome. The top 3 performing planners for each row are highlighted, with the best planner in bold.

| **Seen/small size** (40) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | 22 | **40** | 16 | 33 | 30 | 39 | **40** | **40** | **40** | 36 | 37 | 39 | 23 | 29 | 35 | 39 | 37 |
| | lazy | 22 | **40** | 16 | 35 | 33 | **40** | **40** | **40** | **40** | 34 | 34 | **40** | 25 | 32 | 37 | 38 | **40** |
| mean | eager | 22 | **40** | 16 | 33 | 30 | **40** | **40** | **40** | **40** | 38 | 37 | 19 | 29 | 38 | 29 | 39 | **40** |
| | lazy | 22 | **40** | 16 | 35 | 33 | **40** | **40** | **40** | **40** | **40** | 39 | 23 | 35 | 39 | 34 | 39 | **40** |
| **Unseen/large size** (90) | | | **Baseline** | | | | **Domain-dependent** | | | | | | **Domain-independent** | | | | | |
| **Readout** | **Search** | blind | $h^{\text{gc}}$ | $h^{\max}$ | $h^{\text{add}}$ | $h^{\text{FF}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ | $h^{\text{SDG}}$ | $h^{\text{SDG}^{\text{E}}}$ | $h^{\text{FDG}}$ | $h^{\text{FDG}^{\text{E}}}$ | $h^{\text{LDG}}$ | $h^{\text{LDG}^{\text{E}}}$ |
| sum | eager | - | 7 | - | - | - | 2 | 8 | - | 2 | 4 | - | 3 | - | 2 | 2 | 7 | **12** |
| | lazy | - | 5 | - | - | - | - | **13** | 3 | 4 | 5 | - | 7 | 1 | 2 | 5 | 7 | 12 |
| mean | eager | - | 7 | - | - | - | 7 | 6 | - | 1 | 6 | 3 | - | - | - | - | **10** | 9 |
| | lazy | - | 5 | - | - | - | 7 | 4 | - | 1 | 3 | 3 | - | - | - | - | **9** | 7 |

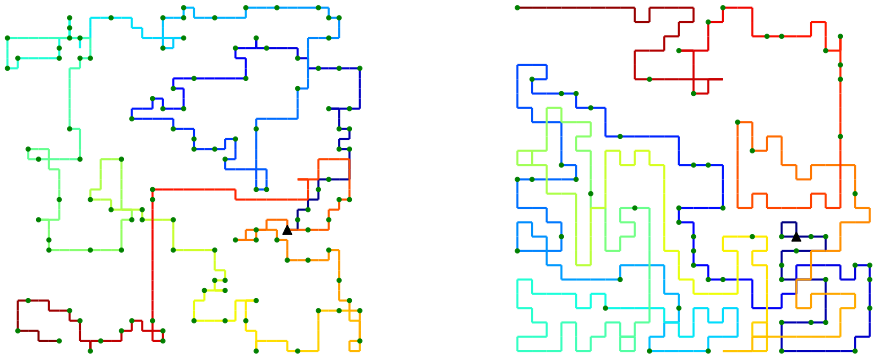
### 7.4.9 VisitSome

In the large problem setting, we train on instances with grid size up to 10, and test on problems with grid size between 15 and 100. Fig. 7.23 and 7.24 illustrate the cumulative coverage of solvers over number of expansions for problems of seen/small size and unseen/large size respectively. Tab. 7.13 provides the coverage table. We refer to Sec. C.5.9 in the appendix for additional cumulative coverage plots of solvers over runtime and plan cost.

VisitSome is a more difficult variant of VisitAll to solve optimally, and also for the goal count heuristic to solve in general as it is forced to search more states. We observe similar patterns from VisitAll where in the domain-dependent trained setting, STRIPS graphs perform best, followed by lifted graphs and lastly by FDR graphs. GOOSE with $\text{SDG}^{\text{E}}$, sum readout and lazy search is able to solve instance `n35-s3` which consists of 1225 grid locations and 119 goal locations with a plan cost of 437 and initial heuristic estimate of 348. Fig. 7.22a illustrates the computed plan which consists of some unnecessary movements such as with the orange section of the path. This may be due to inaccuracies of heuristic estimates during lazy search.

The optimal heuristic for VisitSome is harder to learn than VisitAll, as a necessary component is computing distances from the current state to goal locations. This is not possible for larger problems using GOOSE as the computability of such distances is limited by its receptive field and number of message passing layers.

Interestingly, we observe that domain-independent trained heuristics on lifted graphs perform best on almost all configurations of readout and search algorithms. One possible explanation for this is that although we cannot learn the optimal heuristic due to limited number of message passing layers, we can learn other methods to approximate it which was exposed by training data outside of VisitSome. Domain-independent trained GOOSE with $\text{LDG}^{\text{E}}$, sum readout and eager search is able to solve instance `n25-s0` which



(a) Plan returned by domain-dependent trained GOOSE with $\mathsf{SDG}^{\mathsf{E}}$, sum readout and lazy search for `n35-s3`.

(b) Plan returned by domain-independent trained GOOSE with $\mathsf{LDG}^{\mathsf{E}}$, sum readout and eager search for `n25-s0`.

Figure 7.22: Visualisations of plans returned by GOOSE on VisitSome. The black circle is the initial location, the green circles the goal locations, and the plan starts from dark blue and ends at dark red.

consists of 625 grid locations and 62 goal locations with a plan cost of 447 and initial heuristic estimate of 279. Fig. 7.22b illustrates the computed plan. We notice that the plan makes many unnecessary actions and traverses a majority of the grid. This may be due to an inaccurate heuristic leading the planner to incorrect search regions which it commits to. We recall that the domain-independent training data does not contain any problems from VisitAll which is a similar domain to VisitSome such that GOOSE is not using knowledge from similar domains. We lastly note that the search space is exponential in the number of grid locations.

In the seen and small size instance case, all domain-dependent trained GOOSE models expand fewer nodes than classical heuristics which suggests strong generalisation to problems with sizes seen during training.

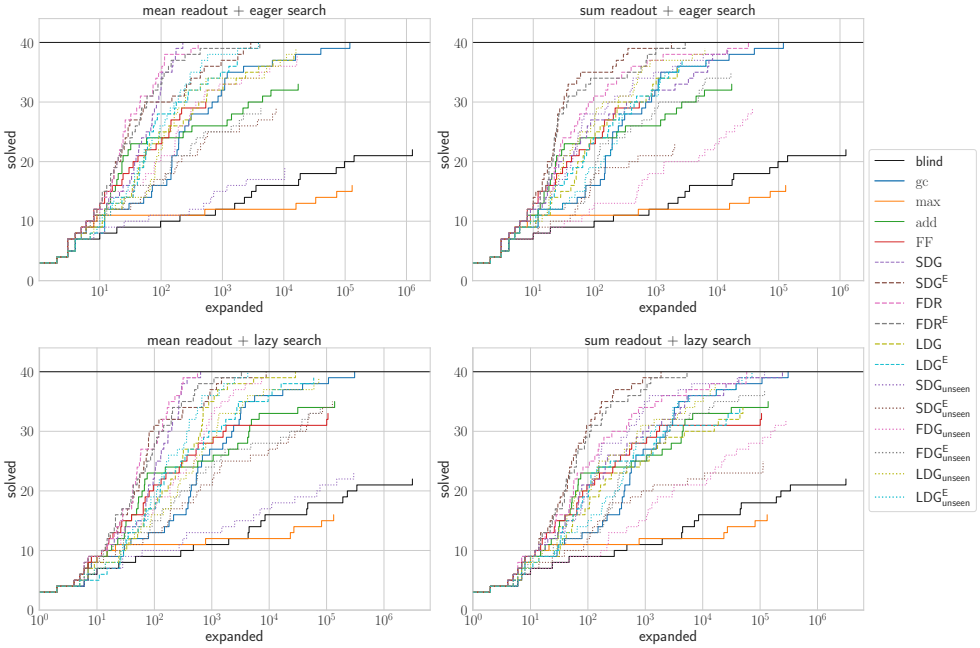



Figure 7.23: Cumulative coverage over number of expanded states on seen/small size VisitSome instances. Total number of problems: 40.

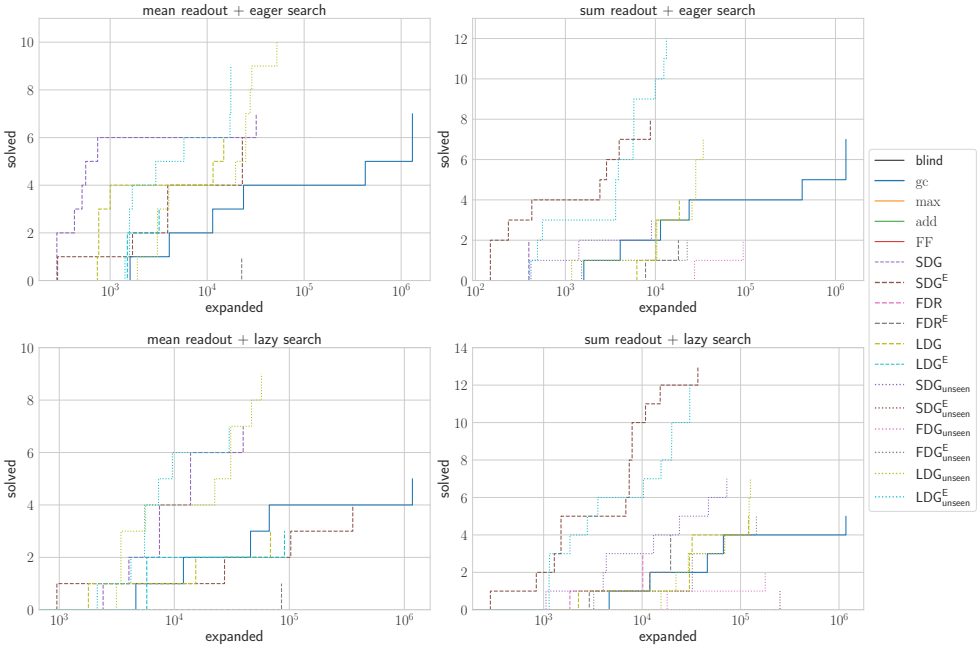



Figure 7.24: Cumulative coverage over number of expanded states on unseen/large size VisitSome instances. Total number of problems: 90.

## 7.5 CPU vs GPU runtime

In this section we comment on the speedups gained by utilising GPUs for heuristic evaluations. We measured the time to evaluate the heuristic value of the initial state represented by FDG and LDG on a single AMD EPYC 7282 2.8GHz CPU core. We also measured the time to evaluate a batch of $n$ copies of the initial state with $n \in \{1, 2, 4, 8, 16, 32, 64, 128\}$ on a single NVIDIA GeForce RTX 3090 GPU, including the time for transferring memory between the host (CPU) and device (GPU).

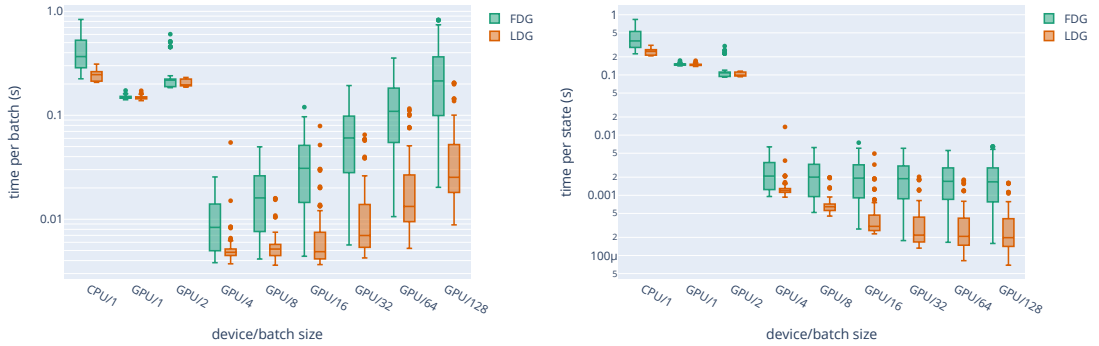



Figure 7.25: Distributions of heuristic evaluation time per batch (left) and per sample (right) on FDG and LDG on the CPU host and GPU device with various batch sizes.

The time spent on heuristic evaluation with multiple configurations of hardware, batch size and graph representations excluding time spent on memory transfers between the CPU host and GPU device is illustrated in Fig. 7.25. We provide both the time for evaluating various batch sizes and the averaged time for evaluating a single state within a batch. We first note that heuristic evaluation of a single state on the GPU is always faster than that on the CPU. However, we should note that the results may change depending on the hardware setup and exclude the overhead of memory transfer.

Furthermore, speedups against the CPU are significantly noticeable once batch sizes are greater than $n = 4$. A possible explanation for this is that the underlying PyTorch Geometric CUDA[3] code is optimised for larger data. For lifted graphs, the speedup by increasing batch size when considering evaluation time per state levels out at around $n = 32$. This may be due to data becoming large such that memory operations within the GPU become the bottleneck for parallelisation. The same explanation can also be applied to the grounded graphs which are much larger and do not exhibit a larger speedup for batch sizes greater than $n = 4$. This also supports our argument in Sec. 6.2 that we should only batch the evaluation of states during heuristic search which are useful, given that the effectiveness of batching evaluation decays with increasing batch

[3] An API for programming for NVIDIA GPUs.

size. We also note that with the optimal batch sizes, lifted graphs are almost an order of magnitude faster to evaluate than grounded graphs. This is not surprising given that they are also significantly smaller graphs as seen in Fig. A.1 in the appendix.

Fig. 7.26 illustrates the ratio of time spent on transferring memory between the CPU host and GPU device to total heuristic evaluation and memory transfer runtime. We note that for small batch sizes of 1 and 2, the transfer is minimal. However, for a majority of cases with larger batch sizes, the ratio of time spent on memory transfers between devices lies between 3% to 10% and decreases with larger batch sizes. This suggests that the underlying implementation may be pipelining memory transfer and computation.

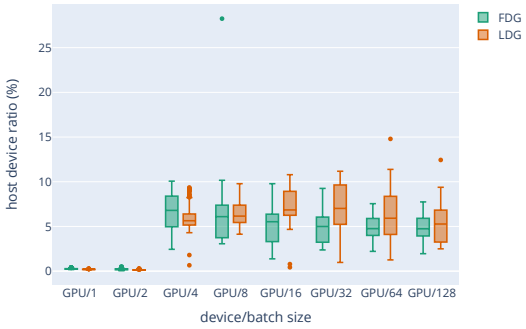



Figure 7.26: Distributions of ratio of time spent on memory transfers between CPU host and GPU device to total heuristic evaluation and memory transfer runtime on FDG and LDG with various batch sizes. The CPU/1 configuration is omitted as it transfers no data to the GPU.

### Evaluation with different GPUs

We also compare evaluation times with different GPUs. The motivation for doing so is that GPU hardware is still improving consistently at the time of writing this thesis, in contrast to sequential processor hardware which has stagnated. Thus, GOOSE is able to achieve passive performance increases due to hardware advancements over time in contrast to classical planners. This heuristic evaluation time is performed on RTX 2080 Ti and RTX A6000 GPUs on a cluster with the same method described previously. Heuristic evaluation results are reported in Fig. 7.27 and key GPU hardware statistics in Tab. 7.14.

We notice that more recent GPUs offer speedups against older GPUs. In lower batch sizes with $n \leq 4$, the 3090 GPU offers a $2\times$ speedup against the 2080 Ti GPUs. The speedup from using the A6000 GPU over the 3090 GPU is still noticeable but less significant for larger data. This is because the A6000 was designed to have more memory for large scale applications rather than improved runtime performance. A more suitable GPU for

Table 7.14: Summary of GPU hardware statistics.

| | RTX 2080 Ti | RTX 3090 | RTX A6000 |
|---|---|---|---|
| Release year | 2018 | 2020 | 2022 |
| Number of cores | 4352 | 10496 | 10752 |
| Boost clock (MHz) | 1545 | 1695 | 1800 |
| Bandwidth (GB/s) | 616.0 | 936.2 | 768.0 |
| Memory size (GB) | 11 | 24 | 48 |

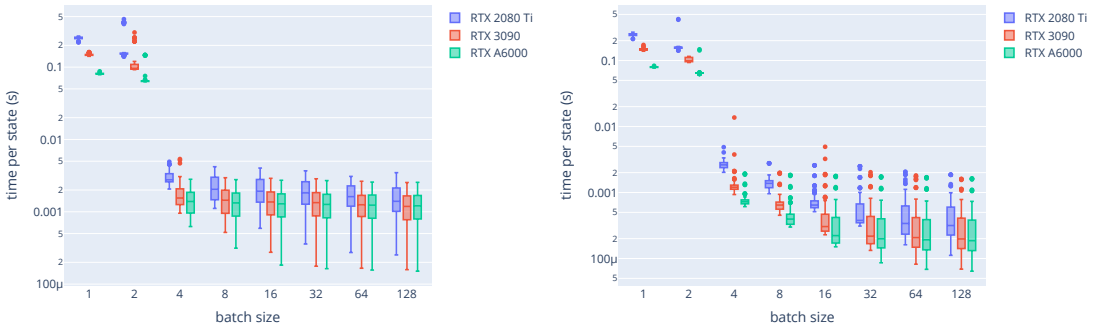



Figure 7.27: Distributions of heuristic evaluation time per sample on FDG (left) and LDG (right) with different batch sizes and GPUs.

comparison against the 3090 GPU from the same year of the A6000 would be the 4090 GPU which has significantly more cores and higher boost clock.

In all comparisons, we observe that the speedups gained from newer versions of hardware are more significant for smaller data such as with LDG and smaller batch sizes. The issue is most likely due to unoptimised memory operations as discussed previously and may alleviated by implementing CUDA code customised for our application. Furthermore, graph neural networks have more unstable parallelisation costs due to sparse matrix multiplication corresponding to message passing updates on graphs. This is in comparison to feed forward or convolutional neural networks for image processing which have more structured computation and hence can more efficiently utilise GPU hardware.

Chapter 8

# Related work

The field of learning for planning is growing increasingly quickly due to the large scale advancements in deep learning approaches such as neural network architectures and deep learning accelerator hardware such as graphics processing units (GPUs) and tensor processing units (TPUs). Nevertheless, to the best of our knowledge, our GOOSE model introduces several novel contributions. It is the first approach for learning domain-independent heuristics from scratch with no assumptions on our domains and provides competitive performance over classical heuristics in both domain-dependent and domain-independent training settings. Another novel contribution is learning domain-independent heuristics using graph representations based on only the lifted planning representation. On top of its strong performance, we provide extensive theoretical and empirical studies on GOOSE's capabilities and limitations of learning domain-independent heuristics.

To further emphasise the novelty of our contributions, it is imperative to explore the large body of related work of learning for planing. We will categorise such prior work into three classes: learning heuristics in Sec. 8.1, learning policies in Sec. 8.2 and other applications of learning for planning in Sec. 8.3. Studies on learning heuristics are most similar to our work. Works on learning policies are worth considering given that there are several strong connections between the heuristic space and the action space. For example, the optimal heuristic $h^*$ encloses all optimal policies. Other works that are not categorised into learning heuristics or policies are also worth considering to appreciate the expanse and impact of this field beyond learning heuristics and policies.

## 8.1 Learning heuristics for planning

The method of learning heuristics for search is not a new concept for GOOSE. However, almost all previous works focus on learning domain-dependent heuristics. We recall from

Sec. 2.2.4 that domain-dependent heuristics are learned from a specific domain and are effective only for that specific domain. We proceed to explore prior work in historical order of when they were published.

Neural network learned heuristics were introduced as early in 2004 by Ernandes and Gori for solving the $n$-puzzle problem with $n^2$ size feature representations of states where the $(n \times k + t)$-th entry is set to 1 if square $k$ is occupied by tile number $t$ and 0 otherwise. Yoon et al. [2008] learns domain-dependent heuristics by learning $\Delta(s) := h^*(s) - h^{\text{FF}}(s)$ for every state with weighted linear combinations of features extracted from the relaxed plan in the computation of $h^{\text{FF}}$. Samadi et al. [2008] represent states as vectors of computed heuristics and use neural networks to learn the optimal heuristic for each domain. They further modify the mean squared error loss function to guide their models to learn admissible heuristics. Jabbari Arfaee et al. [2011] employs bootstrapping by starting with a weak heuristic $h_0$ to provide training samples to learn a stronger heuristic $h_1$ and repeats this bootstrapping process of alternating between generating more useful training samples and learning heuristics from such samples. Garrett et al. [2016] frame learning heuristics for GBFS as a ranking problem and use Rank Support Vector Machines as their learning architecture. Gomoluch et al. [2017] is the first work which learns domain-independent heuristics with neural networks where states are represented by hand engineered features of planning task and values of the $h^{\text{FF}}$ and $h^{\text{cea}}$ [Helmert and Geffner, 2008] heuristics. However, the performance gain is marginal compared to the performance of the $h^{\text{FF}}$ heuristic it leveraged and evaluation was done on few domains and problems. Our work differs in that we do not compute other domain-independent heuristics as features and are still competitive with such heuristics. Francès et al. [2019] use mixed linear programs to learn potential heuristics [Pommerening et al., 2015] for various domains which are descending and dead-end avoiding [Seipp et al., 2016] and thus guaranteed to find a plan in polynomial time with greedy search.

STRIPS-HGN [Shen et al., 2020] is the first model which is able to learn domain-independent heuristics from scratch and was able to achieve informedness better than $h^{\max}$ on certain domains. Its performance may be improved if it has access to more training time and data. However, as discussed in Ch. 3, STRIPS-HGN has three main drawbacks when it comes to learning domain-independent heuristics: (1) its hypergraph representation of planning instances ignores delete lists and thus cannot theoretically learn how to compute $h^*$, (2) its aggregation and update function is not permutation invariant due to the need to order the set of neighbours of each node which may cause it to learn biases in the training data and prevent it from generalising effectively, and (3) it assumes an upper limit on the sizes of action precondition and effects, meaning that in practice it also has to discard certain edges in its hypergraph in its message updating step. We note that STRIPS-HGN can be made permutation invariant in expectation if the order for the set of neighbours of each node is defined randomly. It is also worth considering as future work to fairly compare GOOSE to STRIPS-HGN in the domain-independent training setting given enough computing resources.

A large scale empirical study of domain-dependent neural network heuristics for prob-

lems of same size and domain was conducted to study what deep learning tools were helpful for learning heuristics [Ferber et al., 2020]. States are represented as binary vectors encoding which non-static propositions are true in the state. The study conducted experiments on various neural network parameters such as number of layers, dropout rate of neurons during training and regularisation weight, and also on data size and sampling methods. Neural logic machines (NLMs) [Dong et al., 2019] are neural network architectures that can learn first order logic rules and are applied to learn heuristics [Gehring et al., 2022] from scratch or with the aid of $h^{\text{FF}}$ values as reward generators in the context of reinforcement learning. However, the learned heuristics from scratch are not competitive with even $h^{\text{add}}$ and it is not clear whether leveraging $h^{\text{FF}}$ values provide a speedup over search with $h^{\text{FF}}$ by itself as evaluation is not done by runtime but by termination after 100000 node evaluations. Furthermore, NLMs are domain-dependent architectures as they assume a maximum arity of input predicates. Nevertheless the original NLM paper showed that NLM can learn to generalise and solve Blocksworld instances, sort arrays, and solve path finding problems with perfect accuracy.

Ferber et al. [2022] also provides a comprehensive study of other neural network heuristics which study three methods for learning heuristics: with bootstrapping using different state representations and deeper neural networks than Jabbari Arfaee et al. [2011], with bootstrapping but learning an estimator of search space size instead of a heuristic, and learning from approximate value iteration values. Bootstrapping can also be generalised to leapfrogging which is the same as bootstrapping except that the learned heuristics can generalise to planning instances with different sizes and number of objects, contrary to work by Jabbari Arfaee et al. [2011] and Ferber et al. [2022]. Leapfrogging was explored in detail by Karia and Srivastava [2021] which learns both heuristics and actions at the same time using neural networks. The neural network architecture makes three classes of predictions: a heuristic prediction, an action schema prediction, and predictions of the parameters of the action schema in order to produce a ground action.

Lastly, Staahlberg et al. [2022b] provide a theoretical analysis of which domains MPNNs can learn the optimal heuristic for by using the well known result concerning the connection between MPNNs and 2-variable counting logics [Cai et al., 1992, Xu et al., 2019, Barceló et al., 2020]. Similar to how our work, no theoretical guarantees concerning generalisation are provided. Instead generalisation is empirically evaluated and it shows that the learned heuristics in conjunction with A$^*$ search provides optimal plans for domains whose optimal heuristic value can be represented by 2-variable counting logics. We note that their methods are inherently domain-dependent given that their models use different MLP update functions for predicates of the planning problems. This means that they cannot generalise to unseen problems with a different number of predicates. However, we reiterate that the work was developed for a constructive theoretical analysis of MPNNs for domain-dependent planning instead of constructing fast performing models or domain-independent learning.

## 8.2 Learning generalised policies for planning

Given a planning task with state space $S$ and set of actions $A$, a policy is a function $\pi : S \to A$ that maps a state to an action to take. Policies are usually only considered in planning under uncertainty where we have to account for multiple effects of stochastic actions, meaning that a plan, i.e. a sequence of actions, does not guarantee us of reaching the goal with probability 1. However, it is easy to see that a policy is a generalisation of a plan. Generalised policies are policies that are defined on states for multiple planning tasks, for example the set of all possibly infinitely many tasks of the Blocksworld domain.

Before exploring prior work for learning generalised policies, we note that it is possible to generate a policy from a heuristic function. For classical planning, this is done for each state $s$ by selecting the action $a$ corresponding to the best successor state $s'$ with the lowest heuristic value estimate. This is indeed analogous to choosing greedy actions with the best $Q$-value in the context of probabilistic planning or reinforcement learning, where $Q(s, a)$ provides an estimate of the expected cost to a goal when executing action $a$ in the state $s$. We also discuss a possible method for learning domain-independent policies with our GOOSE method later in Sec. 9.3.1.

We primarily focus on neural network architectures for learning generalised policies for planning beginning with ASNets [Toyer et al., 2020] and refer to Sec. 7 of the same paper for a comprehensive literature review of generalised policy learning methods before the ASNets era. ASNets can be viewed as a graph neural network architecture with message passing performed between grounded propositions and actions, with different update functions corresponding to different action schema. Concurrent with the original ASNets paper [Toyer et al., 2018] were handcrafted convolutional neural network and graph neural network architectures for learning policies for Sokoban and the travelling salesman problem (TSP) respectively [Groshev et al., 2018]. ASNets have also been combined with Upper Confidence Bounds applied to Trees (UCT) [Shen et al., 2019] to aid improper policies returned from ASNets, resulting from suboptimally trained ASNets models or problems that are too hard for ASNets to learn completely. Staahlberg et al. [2022a] also leverage their GNN architecture for learning 2-variable counting logics to learn generalised policies in an unsupervised fashion by using a modified Bellman error loss function.

There also exists a branch of works focusing on learning generalised policies for probabilistic planning expressed in RDDL. All such methods leverage graph neural networks for domain-dependent learning. ToRPIDo [Bajpai et al., 2018] represent states with graphs where nodes represent state variables and edges are constructed for between nodes whose corresponding objects are connected by non-fluents in the domain. Thus, action information is implicitly learnt during training. Furthermore, the action state decoder is fixed meaning that policies only transfer between problems of the same size. TraPSNet [Garg et al., 2019] improves on ToRPIDo by learning generalised policies for problems of variable size. However, the model assumes that there is one binary non-fluent $nf$ and all other fluents, non-fluents and action templates are unary. Their

graph representation consists of object nodes, and edges between two objects $o$ and $o'$ if $nf(o, o') = 1$. SymNet [Garg et al., 2020] and SymNet2 [Sharma et al., 2022] are also generalised policy learners for RDDL which do not have the restrictions of TraPSNet and ToRPIDo. SymNet constructs $|A| + 1$ disjoint graphs, where $|A|$ denotes the number of actions. Each graph have the same set of nodes given by tuples of objects, with differences in their edges. SymNet2 extends SymNet to be more expressive by noticing that certain pairs of actions are scored the same in SymNet under certain conditions.

## 8.3 Other applications of learning for planning

There are also several other approaches for applying learning for planning, either to aid with the solving process as has done by learning heuristics and generalised policies, but also to aid with modelling planning problems themselves.

### Other methods to aid solving with learning

A simple usage of learning for planning is by performing model selection from a portfolio of classical planners. Such methods are known as portfolio-based methods where we have a learner which predicts for each planning problem what is the fastest planner to solve that problem. The motivation for such a technique is that there no is planner that dominates all other planners on all domains. Delfi [Katz et al., 2018] is one such method which employs a convolutional neural network (CNN) for learning the best planner for each planning task. They represent tasks as images by using the abstract structure graph (ASG) representation of the graph which they turn into an image by using its adjacency matrix. Ma et al. [2020] uses GNNs on the FDR problem description graph (FDG) and ASG representations of planning tasks and also introduces adaptive scheduling by changing the chosen planner to a second one midtask if the first does not solve the task in a given time limit. However, the effective of such methods rely on progress of classical planners.

Learning can also be used to provide transformations of our planning task to make it easier to solve when given to another planner. Gnad et al. [2019] uses various classical machine learning approaches such as decision trees, logistic regression, kernel ridge regression, linear regression, and support vector machine regression in order to learn which parts of a problem to ground for a planning task. The motivation is that some problems are infeasible to completely ground but it is possible to solve a partially grounded version of the problem. The method includes learning a priority score for actions during a modified version of the grounding algorithm of Fast Downward [Helmert, 2006]. To enforce partial grounding, the algorithm is terminated after a specified upper bound of actions have been created. Given that this may lead to unsolvable problems, the method involves incremental grounding of more actions when partially grounded tasks are found unsolvable.

PLOI [Silver et al., 2021] takes an alternative approach and learns which subsets of

objects are sufficient for solving the planning problem, motivated by the idea that not all objects in a task are useful for solving the task. The authors use a GNN to score the usefulness of objects for solving a task, and incrementally plan with larger subsets of objects until a plan is found. Their model scales well over the base planner they use on problems with large number of objects but few goals. They do note that the effectiveness of their method depends on the relative number of necessary objects to all objects in the problem, where if all objects in the task are necessary PLOI has no impact.

### Methods to aid modelling with learning

Another bottleneck in using model-based planning besides the solving process is the modelling process. Modelling is usually done by humans handwriting domains and problems in PDDL which may be prone to errors and may not account for unseen obstacles in the real world. Despite its lack of robustness to unseen instances, learning models are usually good at handling noise common in the real world. This idea has been leveraged for automatically learning safe planning models from plans and trajectories [Stern and Juba, 2017, Juba et al., 2021] with extensions to deal with stochastic [Juba and Stern, 2022] and numeric planning [Mordoch et al., 2023].

Bonet and Geffner [2020] learn planning models from the structure of the state space with no additional information with encodings of SAT theories. The theories are specified by a set of hyperparameters representing upper bounds on the number of action schemas, predicates and arguments of the learned planning models. In this way the problem is phrased as a search over hyperparameters which return the best satisfiable theories corresponding to the simplest planning models. We also refer to [Arora et al., 2018] for a comprehensive review of other classical machine learning methods for learning planning models.

Chapter 9

# Conclusion

We conclude our work in this chapter by providing a summary of our major contributions and questions we aimed to answer in this thesis, a transparent discussion about current limitations of our work, and a large trove of promising future work to be explored.

## 9.1 Contributions

The main focus of our thesis was to speed up solving planning problems by leveraging major advances in deep learning algorithms and hardware. Our contributions span across various topics and themes.

### Modelling

We developed novel graph representations of planning tasks with the goal of learning domain-independent heuristics when combined with graph representation learning models such as message passing neural networks (MPNNs). The graphs include grounded and lifted variants, both with their own advantages and disadvantages. Namely, our grounded graph representations $\mathsf{SDG}^{\mathsf{E}}$ and $\mathsf{FDG}^{\mathsf{E}}$ provide more expressivity when used in conjunction with MPNNs, while our lifted graph representations $\mathsf{LDG}$ and $\mathsf{LDG}^{\mathsf{E}}$ are much smaller and quicker to evaluate with MPNNs. Our work also provides the first domain-independent graphs with no assumptions on the planning domains we work with.

### Theory

We also provided a comprehensive theoretical and empirical study to help us answer the question *what can we learn?* More specifically, we identify the domain-independent heuristics that we can or can not learn with our graph representations and MPNNs. We also provide a discussion on the impacts of our results and complement the theory with a set of experimental results.

### Parallelisation

We introduced more intelligent algorithms and ideas for parallelising heuristic evaluations with GPUs during heuristic search in order to speed up our planner. This is done by observing that naive methods of batching as many heuristic evaluations as we can is not 'useful', given that there are nontrivial parallelisation costs and not all heuristic evaluations are necessary for the search algorithm. Motivated by this, we introduced ideas of adaptive batching of heuristic evaluations which observe the local topology of the search space in order to increase or decrease parallelisation of heuristic evaluation. Our parallelised algorithms can also be leveraged by classical multicore and multithread processors.

### Learning for planning

Our final contribution is the culmination of all our novel learning and planning contributions in the form of the GOOSE model for learning to solve planning tasks quickly. Furthermore, we provide a second diverse and comprehensive set of experiments that establish a new standard of empirical evaluation for the field. We have put effort in both the learning and planning sides of our implementation in order to optimise GOOSE to be tested against state-of-the-art planners and be competitive in terms of runtime on certain domains. Our experiments also consist of test problems with number of objects of up to 10 times the number seen in training samples in order to transparently identify the generalisation limits of GOOSE.

## 9.2 Limitations

As with all research works, there exist several possible limitations. One conceptual limitation with this line of research is the expensive evaluation procedure as we required access to many hundreds of GPU hours computing time to test and prototype our models and perform experiments. For example, evaluating GOOSE performance on search alone requires up to 5008 GPU hours: 2 search configurations (eager and lazy) $\times$ 2 training taxonomies (domain-dep and domain-indep.) $\times$ 6 graph representations $\times$ 2 model readout configurations $\times$ 626 test instances $\times$ $\frac{1}{6}$hr timeout. Note that this is a worst case bound since some problems require much less time than our cutoff of 600 seconds to be solved, and we can also terminate evaluation on a domain early when no problems are solved given that problems can be sorted in order of difficulty. Even with just one model configuration, evaluation on all 626 test instances with the given timeout requires at most 100 GPU hours. All of this ignores additional time required for training and prototyping models. This resource issue limits reproducibility of the work but also explains smaller problem sets used in empirical evaluation in the literature, with testing done on problems with lower difficulty and timeouts.

Another current limitation with this work and the current literature with the domain-dependent training setting is its practical usage when compared to classical planners.

For example, it may be the case that we have access to few or no training samples for a real life planning problem and domain. Although this is a motivation for introducing domain-independent learning of heuristics, and we suggest that GOOSE is competitive in learning domain-independent heuristics with respect to other works in the field of learning for planning, its performance is not yet significantly more competitive than classical planners. Nevertheless, this is not a dead end since we can passively rely on advances in GPU hardware, collection of more data, and explore various methods described in the following section to improve GOOSE. Moreover, transductive, problem-specific learning [Ferber et al., 2022] can also complement domain-independent heuristics, where we spend time on learning additional information about each planning problem to aid with search as a preprocessing step. This is not an unreasonable strategy and is also common in classical planning methods such as abstraction heuristics.

Furthermore, there are classes of planning problems such as Hanoi and Sokoban where learning domain-dependent heuristics still perform inferior to classical heuristics in terms of runtime and informedness. One explanation for this is the fixed receptive field of GOOSE which limits model expressivity and is also major limitation for other neural network methods for planning [Toyer et al., 2020, Shen et al., 2020]. Nevertheless, it is worth noting that methods for solving computationally hard problems are generally complementary to one another, meaning that there is no one method that is better than all others for all problems.

## 9.3 Future work

Given the large theoretical and practical scope of our work, there remains a lot of future work to be investigated. These can be categorised into three main components: additional model modifications and engineering to further optimise performance, extending our model to account for more expressive classes of planning, and additional theoretical studies to provide a yardstick for understanding the potentials and limits of this research direction.

### 9.3.1 Improving performance

There are various additional methods to improve the performance of GOOSE, ranging from pure optimisation of the runtime of the code to exploring methods of improving domain-dependent and domain-independent training and learning.

#### Additional engineering effort

On the implementation side, it is possible to further optimise our heuristic evaluations by implementing our entire learning component in C++ or possibly even CUDA code over what we have done which is using Python packages and calling Python code with pybind. Low level optimisations of this form may include reducing the overhead costs of transferring graph data between the CPU host and GPU device on each iteration of

search. On the learning side, we may also generate a greater and more diverse dataset for domain-independent training. On the planning side, we may explore and refine our ideas for adaptive batched search algorithms proposed in Sec. 6.

### Improving training for generalisation

With the exception of Spanner, our model struggles to learn useful heuristics when the sizes of our problems becomes much larger than instances it has been trained on. However, we are still able to achieve significant improvements over classical heuristics on problems with unseen size with a reasonable range. This suggests that methods like leapfrogging [Groshev et al., 2018, Karia and Srivastava, 2021] applied to our model may allow us to continually learn to solve increasingly large planning problems.

Also apparent with other models which exploit learning for planning, there is still no clear robust method of training neural network heuristic functions which can generalise well to unseen problems. Validation techniques alleviate the issue somewhat but have no strong theoretical guarantees. More specifically, there is no method to guarantee that retraining the model using the same methods will provide consistent results with low variance, and no method to predict when we are overfitting to the training set. Lastly, we recall that we have not tuned any hyperparameters of the model nor of the optimiser and doing so may improve on our collected results.

### Learning preferred operators

Both eager and lazy heuristic search methods benefit from preferred operators as they provide an additional layer of informativeness, and empirical evaluation has suggested that they are a very powerful technique for satisficing search. It is possible to learn preferred operators with our graph representations and exploit dual queue search algorithms which use preferred operators. Similarly to ASNets [Toyer et al., 2020], we can learn to perform binary classification on nodes corresponding to applicable grounded actions in order to learn the best possible action we can perform at any state. This would mean our model will output both a heuristic value, and confidence values on applicable node actions for each graph. The training loss would thus be a weighted sum of mean squared error and binary cross entropy. For our lifted graphs, this would require extending their definition in order to represent applicable grounded nodes of a given state. One can in fact also view learning preferred operators described this way as learning *domain-independent generalised policies*.

### Intelligent domain-independent training paradigms

It is possible to leverage ideas from zero-shot or transductive learning in order to learn from test instances without true labels in order to improve on a pretrained model with no knowledge of the task it is solving. However, such techniques are usually of a statistical nature which have certain assumptions of the distributions of the data it is performing

inference for and may be difficult to directly apply to planning tasks which may have more complex distributions due to their inherent computational hardness.

Furthermore, it is worth experimenting with domain-dependent training of GOOSE using pretrained domain-independent parameters.

#### Graph representation improvements

We noted that in some experiments, the lifted graphs have higher generalisation and informativeness capabilities than grounded graphs. One possible reason for this is that their compact nature provides a form of regularisation. A more compelling reason is that they encode information about predicates and action schema where this information is lost for grounded graphs, and MPNNs may not be able to recover the predicates and action schema from the structure of grounded graphs due to their expressivity limits. Thus, it is possible to introduce predicate and schema information into our grounded graphs in a similar fashion to what is done with our lifted graphs.

It is also possible to modify certain aspects of our novel lifted graphs, LDG and $\mathsf{LDG}^{\mathsf{E}}$. For example, the current representation has a disjoint union of goals and facts true in the given state as done with ASGs but it is also possible to take the union instead. We may also investigate other forms of positional encoding PE functions for encoding argument indices.

### 9.3.2 Extensions for more expressive planning

For our work, we have been mainly focused on learning to solve classical planning problems, i.e. we assume that the world is static, discrete and deterministic. However, there exist a myriad of more expressive planning models with which it is possible to leverage learning to solve. As suggested by Shen [2019] we can learn heuristics for probabilistic planning problems by constructing graph representations of Stochastic Shortest Path problems (SSP) [Bertsekas and Tsitsiklis, 1991]. For ground graphs, this would mean introducing additional nodes and edges to represent probabilistic effects of stochastic actions. One natural extension of $\mathsf{SDG}^{\mathsf{E}}$ is illustrated in Fig. 9.1. For lifted graphs, an extension may introducing additional nodes and edges of the graph layers described in Fig. 3.5b corresponding to action schema to encode probabilistic schema effects.

Numeric planning, introduced in PDDL2.1 [Fox and Long, 2003], consists of numeric states where state variables may have infinite domains given by rational numbers, and preconditions, effects and optimisation functions given by linear equations of numeric variables. We can encode numeric planning problems by introducing additional nodes with numeric features beyond one hot encodings corresponding to numeric variables, action preconditions and effects, and optimisation functions. It is easy to adapt heuristic search to numeric planning given that it has well defined state successor generators.

We may also learn heuristics for multi-objective planning in which heuristics are now *sets of vectors* instead of scalars [Geißer et al., 2022, Chen et al., 2023]. One possible way

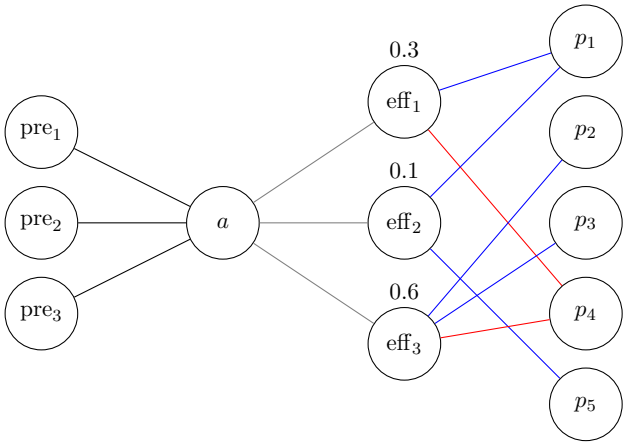



Figure 9.1: A possible $\mathsf{SDG}^{\mathsf{E}}$ extension to deal with stochastic planning. The figure illustrates the subgraph of an action $a$ with $\text{pre}(a) = \{\text{pre}_1, \text{pre}_2, \text{pre}_3\}$, and three probabilistic effects of the form (prob. of activating, add, del) with $\text{eff}_1(a) = (0.3, \{p_1\}, \{p_4\})$, $\text{eff}_2(a) = (0.1, \{p_1, p_5\}, \emptyset)$, and $\text{eff}_3(a) = (0.6, \{p_2, p_3\}, \{p_4\})$.

to extend our methods to learn multi-objective heuristics is to modify our graph neural networks into generative models from which we sample learned joint distributions of our input and output data. For example, given an input graph representation of our planning task, our model may learn to return an associated Gaussian Mixture Model where means of Gaussian components correspond to heuristic vectors in our sets. Indeed, neural networks have been employed to learn mixture models [Bishop, 1994]. However, learners in this direction are unlikely to be completely domain-independent given that the output heuristic vector sizes are dependent on the domain and the models we consider cannot learn to output variable sized vectors. Furthermore, by learning to output vectors instead of scalars, we may lose permutation invariance of multiple objectives across domains.

### 9.3.3 Open theoretical questions

As discussed in Sec. 4.3.1, there are multiple directions we can take to provide more informed theoretical results. One direction is to further refine our expressivity results by considering classes of planning tasks where we can learn $h^*$ with our graph representations and MPNNs, and the probabilities of doing so. We may also consider more expressive GNN models which are still tractable for our setting. Another direction is to study generalisation bounds or extrapolation results of our learned models. A more impactful direction may be to provide general theory for learning for planning which is agnostic to the choice of (tractable) learning models such as GOOSE, STRIPS-HGN or ASNets.

## 9.4 Final remarks

Model-free learners and model-based solvers are the two major paradigms of artificial intelligence in the current age. Due to large scale advances and interest of deep learning which are also generally favoured by political influences and the age of information, it is inevitable that we can no longer ignore model-free learners and conversely we can no longer focus our attention solely on model-based solvers.

In this thesis we contribute to bridging the gap between these two major paradigms. We introduce GOOSE for leveraging several advances in deep learning research and hardware in order to aid with solving combinatorially difficult planning problems more efficiently. Although we have identified several major contributions in our work, there are still many more open problems to solve and various possible research directions to explore as discussed in our extensive future works section.

Appendix A

# Graph and dataset statistics

## A.1 Graph sizes

Fig. A.1 and A.2 illustrate the distributions of graph size of the various graph representations on the dataset described in Ch. 5.

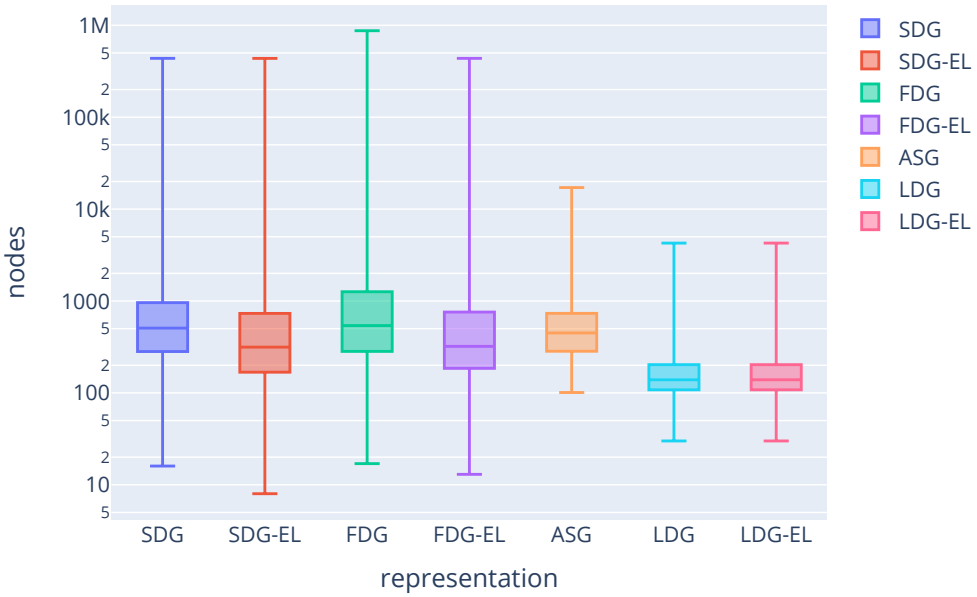



Figure A.1: Box plots of number of nodes for various graph representations.

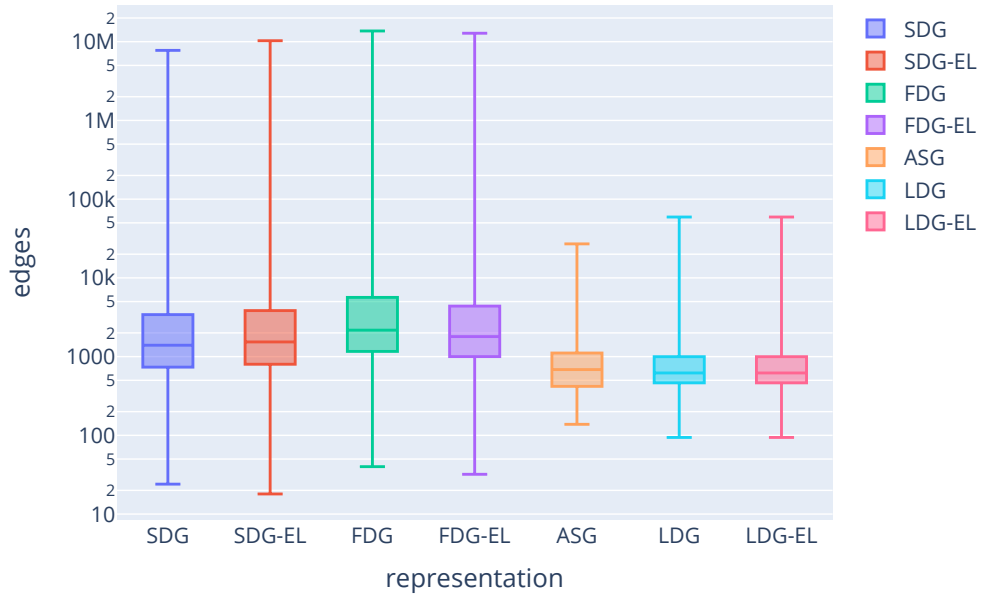



Figure A.2: Box plots of number of edges for various graph representations.

## A.2 Inference dataset information

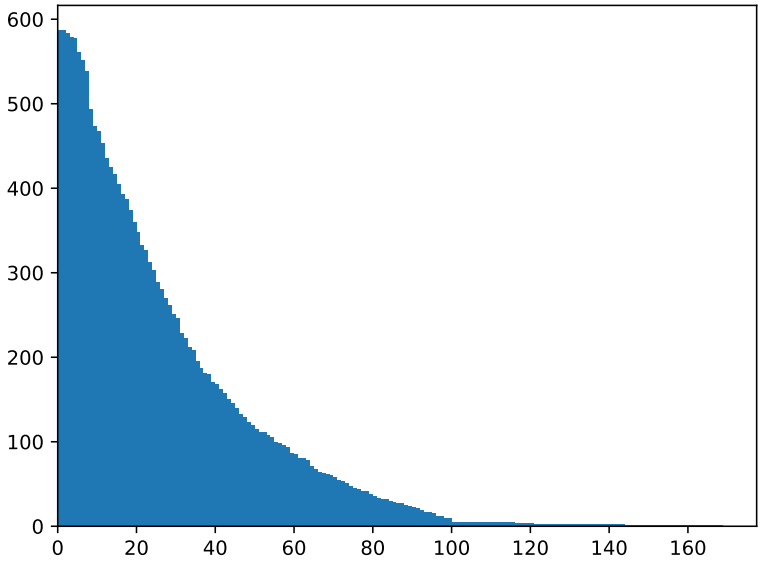



Figure A.3: Distribution of samples by their $h^*$ labels.

Table A.1: Unitary cost domains from 1998-2018 IPCs with corresponding number of solved instances and states for use in the inference experiments described in Ch. 5.

| Domain | # data | # plans |
|---|---|---|
| ipc-1998-grid-2 | 79 | 3 |
| ipc-1998-logistics-1 | 483 | 14 |
| ipc-1998-logistics-2 | 140 | 5 |
| ipc-1998-movie-1 | 240 | 30 |
| ipc-1998-mystery-1 | 153 | 19 |
| ipc-1998-mystery-prime-1 | 238 | 31 |
| ipc-1998-mystery-prime-2 | 37 | 5 |
| ipc-2000-elevator-simple | 7356 | 145 |
| ipc-2000-freecell | 1746 | 50 |
| ipc-2000-logistics | 1492 | 33 |
| ipc-2002-depots | 396 | 14 |
| ipc-2002-driverlog | 312 | 15 |
| ipc-2002-freecell | 800 | 18 |
| ipc-2002-rovers | 301 | 13 |
| ipc-2002-satellite | 203 | 10 |
| ipc-2004-pipesworld-no-tankage | 482 | 25 |
| ipc-2004-pipesworld-tankage | 331 | 18 |
| ipc-2004-satellite | 203 | 10 |
| ipc-2011-no-mystery | 517 | 20 |
| ipc-2011-parking | 168 | 8 |
| ipc-2014-barman | 150 | 3 |
| ipc-2014-hiking | 450 | 18 |
| ipc-2014-parking | 170 | 8 |
| ipc-2018-organic-synthesis | 19 | 7 |

# Appendix B

# Additional results for inference

## B.1 Best performing model scores

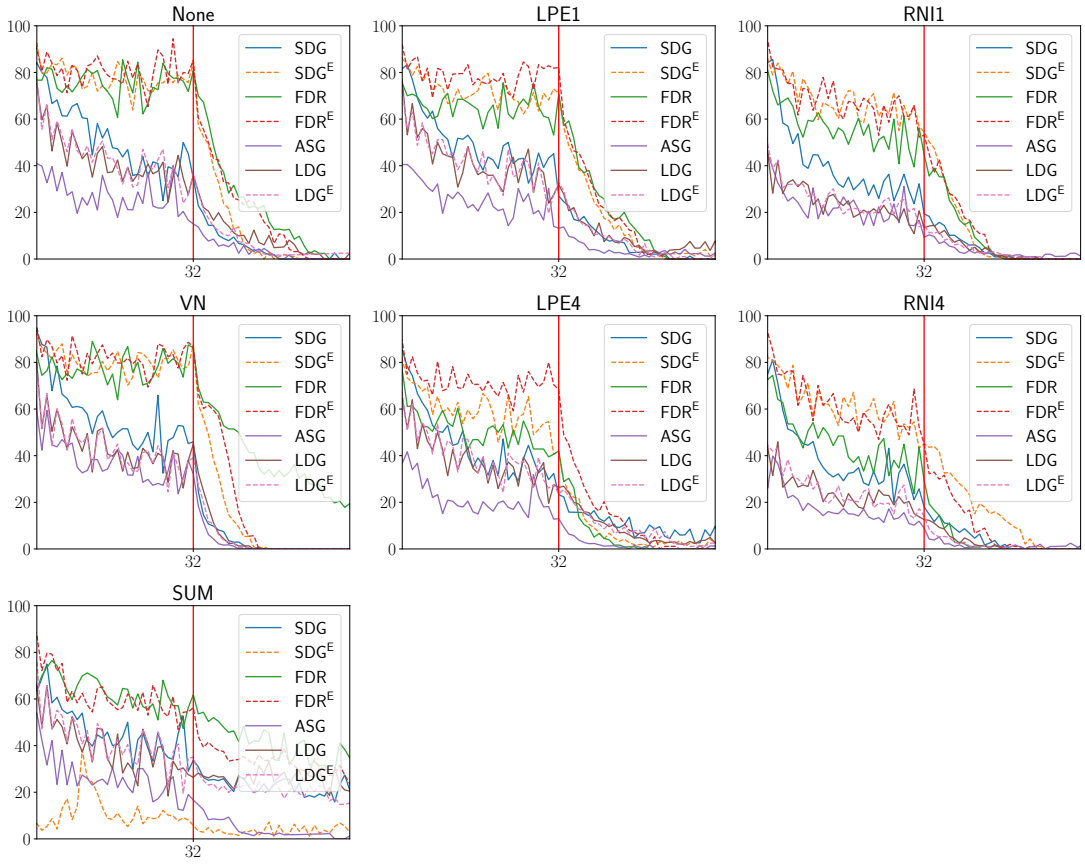



Figure B.1: Maximum accuracy per target $h^*$ value over 5 experiment repeats. The vertical red line indicates the interval of the $h^*$ values which the model was exposed to during training. $y$-axis: accuracy, $x$-axis: target $h^*$ value.

# Appendix C

# Additional results for search

## C.1 Domain-dependent training validation scores

Table C.1: Validation metrics of best model with **sum readout** and **domain-dependent** training chosen for each domain. w. loss represents the best weighted train and validation loss given by Eq. 5.3

| Domain | Metric | SDG | SDG$^{E}$ | FDG | FDG$^{E}$ | LDG | LDG$^{E}$ |
|---|---|---|---|---|---|---|---|
| Blocksworld (3) | w. loss | 0.06 | 0.00 | 0.13 | 0.06 | 0.09 | 0.18 |
| | mean runtime | 2.9 | 7.3 | 4.0 | 9.8 | 6.7 | 25.8 |
| | mean expanded | 30 | 29 | 30 | 30 | 46 | 311 |
| | solved | 3 | 3 | 3 | 3 | 3 | 3 |
| Ferry (3) | w. loss | 0.36 | 0.02 | 0.04 | 0.04 | 6.38 | 7.73 |
| | mean runtime | 6.7 | 16.6 | 7.5 | 18.7 | 41.9 | - |
| | mean expanded | 52 | 38 | 38 | 38 | 976 | - |
| | solved | 3 | 3 | 3 | 3 | 1 | 0 |
| Gripper (1) | w. loss | 0.48 | 0.47 | 0.39 | 0.35 | 0.21 | 0.21 |
| | mean runtime | 3.2 | 6.7 | 3.3 | 10.3 | 6.5 | 12.1 |
| | mean expanded | 35 | 43 | 35 | 38 | 40 | 36 |
| | solved | 1 | 1 | 1 | 1 | 1 | 1 |
| Hanoi (1) | w. loss | 27.72 | 44.16 | 93.98 | 10.47 | 621.35 | 1036.46 |
| | mean runtime | - | - | - | - | - | - |
| | mean expanded | - | - | - | - | - | - |
| | solved | 0 | 0 | 0 | 0 | 0 | 0 |
| $n$-puzzle (3) | w. loss | 0.39 | 0.57 | 2.19 | 1.47 | 169.92 | 170.34 |
| | mean runtime | - | - | - | - | - | - |
| | mean expanded | - | - | - | - | - | - |
| | solved | 0 | 0 | 0 | 0 | 0 | 0 |
| Sokoban (3) | w. loss | 0.11 | 0.02 | 0.03 | 0.06 | 20.73 | 20.66 |
| | mean runtime | 49.7 | 248.3 | 71.9 | 17.7 | 93.9 | 133.2 |
| | mean expanded | 894 | 1372 | 496 | 70 | 4646 | 1630 |
| | solved | 3 | 2 | 2 | 2 | 3 | 2 |
| Spanner (3) | w. loss | 0.00 | 0.00 | 0.12 | 0.00 | 0.25 | 0.26 |
| | mean runtime | - | 14.2 | 4.6 | - | 8.2 | 18.8 |
| | mean expanded | - | 31 | 31 | - | 31 | 31 |
| | solved | 0 | 3 | 1 | 0 | 3 | 3 |
| VisitAll (3) | w. loss | 0.04 | 0.03 | 0.04 | 0.04 | 0.04 | 0.04 |
| | mean runtime | 22.7 | 31.6 | 17.8 | 48.8 | 18.2 | 50.3 |
| | mean expanded | 334 | 222 | 258 | 258 | 328 | 334 |
| | solved | 3 | 3 | 3 | 3 | 3 | 3 |
| VisitSome (3) | w. loss | 6.35 | 0.39 | 4.33 | 1.44 | 6.37 | 6.25 |
| | mean runtime | 4.8 | 10.1 | 6.1 | 135.7 | 85.5 | - |
| | mean expanded | 67 | 56 | 101 | 1016 | 2883 | - |
| | solved | 2 | 3 | 3 | 2 | 2 | 0 |

Table C.2: Validation metrics of best model with **mean readout** and **domain-dependent** training chosen for each domain. w. loss represents the best weighted train and validation loss given by Eq. 5.3

| **Domain** | **Metric** | SDG | SDG$^{E}$ | FDG | FDG$^{E}$ | LDG | LDG$^{E}$ |
|---|---|---|---|---|---|---|---|
| Blocksworld (3) | w. loss | 0.07 | 0.03 | 0.03 | 0.01 | 0.06 | 0.05 |
| | mean runtime | 3.2 | 7.1 | 3.3 | 9.1 | 6.0 | 10.1 |
| | mean expanded | 30 | 29 | 29 | 29 | 32 | 27 |
| | solved | 3 | 3 | 3 | 3 | 3 | 2 |
| Ferry (3) | w. loss | 0.38 | 0.02 | 0.05 | 0.02 | 8.12 | 7.60 |
| | mean runtime | 7.1 | 16.9 | 7.0 | 20.4 | 207.0 | 517.7 |
| | mean expanded | 46 | 39 | 39 | 42 | 3430 | 2691 |
| | solved | 3 | 3 | 3 | 3 | 1 | 1 |
| Gripper (1) | w. loss | 0.34 | 0.25 | 6.37 | 0.72 | 0.21 | 0.23 |
| | mean runtime | 3.4 | 6.9 | 8.9 | 12.0 | 7.8 | 13.7 |
| | mean expanded | 57 | 42 | 137 | 74 | 40 | 38 |
| | solved | 1 | 1 | 1 | 1 | 1 | 1 |
| Hanoi (1) | w. loss | 48.81 | 25.75 | 85.65 | 9.89 | 11513.92 | 320.39 |
| | mean runtime | - | - | - | - | - | - |
| | mean expanded | - | - | - | - | - | - |
| | solved | 0 | 0 | 0 | 0 | 0 | 0 |
| $n$-puzzle (3) | w. loss | 0.41 | 0.22 | 0.95 | 0.29 | 170.14 | 170.20 |
| | mean runtime | 516.1 | - | 257.1 | - | - | - |
| | mean expanded | 3292 | - | 1261 | - | - | - |
| | solved | 1 | 0 | 1 | 0 | 0 | 0 |
| Sokoban (3) | w. loss | 0.04 | 0.02 | 0.03 | 0.02 | 20.71 | 20.67 |
| | mean runtime | 178.0 | 152.7 | 7.7 | 159.4 | 101.1 | 137.7 |
| | mean expanded | 3379 | 833 | 59 | 653 | 4662 | 1634 |
| | solved | 3 | 3 | 3 | 3 | 3 | 2 |
| Spanner (3) | w. loss | 0.00 | 0.00 | 0.00 | 0.00 | 0.23 | 0.22 |
| | mean runtime | - | - | - | - | 8.8 | 19.9 |
| | mean expanded | - | - | - | - | 31 | 31 |
| | solved | 0 | 0 | 0 | 0 | 3 | 3 |
| VisitAll (3) | w. loss | 0.09 | 0.07 | 0.03 | 0.02 | 0.05 | 0.07 |
| | mean runtime | 15.0 | 47.3 | 106.9 | 224.2 | 18.8 | 59.4 |
| | mean expanded | 274 | 325 | 1615 | 1291 | 373 | 390 |
| | solved | 3 | 3 | 3 | 3 | 3 | 3 |
| VisitSome (3) | w. loss | 4.96 | 0.44 | 1.08 | 0.32 | 5.86 | 5.53 |
| | mean runtime | 8.9 | 59.8 | 32.1 | 385.4 | 9.6 | 42.7 |
| | mean expanded | 173 | 440 | 626 | 3135 | 188 | 346 |
| | solved | 3 | 3 | 3 | 3 | 3 | 3 |

## C.2 Domain-independent training validation scores

Table C.3: Validation metrics of best model with **sum readout** and **domain-independent** training chosen for each domain. w. loss represents the best weighted train and validation loss given by Eq. 5.3

| Domain | Metric | SDG | SDG$^E$ | FDG | FDG$^E$ | LDG | LDG$^E$ |
|---|---|---|---|---|---|---|---|
| Blocksworld (3) | w. loss | 2.27 | 1.59 | 96.94 | 0.51 | 6.88 | 4.20 |
| | mean runtime | 13.2 | - | 212.0 | 81.1 | - | - |
| | mean expanded | 264 | - | 5325 | 469 | - | - |
| | solved | 3 | 0 | 3 | 3 | 0 | 0 |
| Ferry (3) | w. loss | 31.56 | 1.59 | 25.36 | 0.51 | 6.88 | 4.20 |
| | mean runtime | 14.0 | - | - | 78.4 | - | - |
| | mean expanded | 119 | - | - | 184 | - | - |
| | solved | 3 | 0 | 0 | 3 | 0 | 0 |
| Gripper (1) | w. loss | 24.34 | 1.59 | 25.36 | 0.51 | 8.92 | 6.88 |
| | mean runtime | 4.4 | - | - | 120.6 | 36.5 | 41.4 |
| | mean expanded | 90 | - | - | 874 | 1250 | 332 |
| | solved | 1 | 0 | 0 | 1 | 1 | 1 |
| Hanoi (1) | w. loss | 2.27 | 1.59 | 25.36 | 0.51 | 6.88 | 4.20 |
| | mean runtime | - | - | - | - | - | - |
| | mean expanded | - | - | - | - | - | - |
| | solved | 0 | 0 | 0 | 0 | 0 | 0 |
| $n$-puzzle (3) | w. loss | 2.27 | 1.59 | 25.36 | 0.51 | 6.88 | 4.20 |
| | mean runtime | - | - | - | - | - | - |
| | mean expanded | - | - | - | - | - | - |
| | solved | 0 | 0 | 0 | 0 | 0 | 0 |
| Sokoban (3) | w. loss | 31.56 | 90.17 | 25.36 | 3.72 | 6.88 | 5.79 |
| | mean runtime | 284.0 | 424.2 | - | 836.1 | 87.5 | 360.7 |
| | mean expanded | 5403 | 2315 | - | 3964 | 4666 | 4569 |
| | solved | 3 | 1 | 0 | 1 | 3 | 3 |
| Spanner (3) | w. loss | 2.27 | 1.59 | 25.36 | 0.51 | 6.88 | 4.20 |
| | mean runtime | - | - | - | - | - | - |
| | mean expanded | - | - | - | - | - | - |
| | solved | 0 | 0 | 0 | 0 | 0 | 0 |
| VisitAll (3) | w. loss | 5.53 | 8.23 | 25.36 | 22.75 | 10.89 | 4.20 |
| | mean runtime | 14.6 | 65.6 | - | 51.6 | 22.8 | 53.0 |
| | mean expanded | 266 | 434 | - | 285 | 446 | 334 |
| | solved | 3 | 3 | 0 | 3 | 3 | 3 |
| VisitSome (3) | w. loss | 2.27 | 8.23 | 96.94 | 37.93 | 10.89 | 6.88 |
| | mean runtime | 465.6 | 96.4 | 49.4 | 131.3 | 13.2 | 34.6 |
| | mean expanded | 9386 | 696 | 979 | 999 | 344 | 266 |
| | solved | 3 | 2 | 1 | 3 | 3 | 3 |

Table C.4: Validation metrics of best model with **mean readout** and **domain-independent** training chosen for each domain. w. loss represents the best weighted train and validation loss given by Eq. 5.3

| Domain | Metric | SDG | SDG$^E$ | FDG | FDG$^E$ | LDG | LDG$^E$ |
|---|---|---|---|---|---|---|---|
| Blocksworld (3) | w. loss | 1.45 | 0.32 | 0.58 | 0.25 | 6.60 | 5.86 |
| | mean runtime | 20.5 | 142.5 | 20.2 | 25.0 | 403.7 | 46.6 |
| | mean expanded | 279 | 813 | 229 | 113 | 20055 | 631 |
| | solved | 3 | 3 | 3 | 3 | 1 | 1 |
| Ferry (3) | w. loss | 1.96 | 0.25 | 0.84 | 0.35 | 4.85 | 4.79 |
| | mean runtime | 42.8 | 203.9 | 20.1 | 51.5 | - | - |
| | mean expanded | 406 | 542 | 133 | 118 | - | - |
| | solved | 3 | 3 | 3 | 3 | 0 | 0 |
| Gripper (1) | w. loss | 1.45 | 0.25 | 0.84 | 0.33 | 5.03 | 5.65 |
| | mean runtime | 23.4 | - | 5.4 | 52.9 | 11.7 | 34.3 |
| | mean expanded | 673 | - | 96 | 600 | 319 | 300 |
| | solved | 1 | 0 | 1 | 1 | 1 | 1 |
| Hanoi (1) | w. loss | 1.45 | 0.25 | 0.58 | 0.25 | 4.85 | 4.79 |
| | mean runtime | - | - | - | - | - | - |
| | mean expanded | - | - | - | - | - | - |
| | solved | 0 | 0 | 0 | 0 | 0 | 0 |
| $n$-puzzle (3) | w. loss | 1.45 | 0.25 | 0.58 | 0.25 | 4.85 | 4.79 |
| | mean runtime | - | - | - | - | - | - |
| | mean expanded | - | - | - | - | - | - |
| | solved | 0 | 0 | 0 | 0 | 0 | 0 |
| Sokoban (3) | w. loss | 3.04 | 0.25 | 0.71 | 0.25 | 5.03 | 5.86 |
| | mean runtime | 402.6 | - | 272.6 | - | 86.6 | 368.0 |
| | mean expanded | 7602 | - | 3660 | - | 4648 | 4666 |
| | solved | 3 | 0 | 3 | 0 | 3 | 3 |
| Spanner (3) | w. loss | 1.45 | 0.25 | 0.58 | 0.25 | 4.85 | 4.79 |
| | mean runtime | - | - | - | - | - | - |
| | mean expanded | - | - | - | - | - | - |
| | solved | 0 | 0 | 0 | 0 | 0 | 0 |
| VisitAll (3) | w. loss | 1.45 | 0.25 | 0.58 | 0.25 | 6.60 | 4.79 |
| | mean runtime | - | - | - | - | 19.2 | 53.8 |
| | mean expanded | - | - | - | - | 334 | 334 |
| | solved | 0 | 0 | 0 | 0 | 3 | 3 |
| VisitSome (3) | w. loss | 1.45 | 0.25 | 1.08 | 0.25 | 5.63 | 4.86 |
| | mean runtime | - | - | 449.5 | - | 12.0 | 53.7 |
| | mean expanded | - | - | 8942 | - | 246 | 429 |
| | solved | 0 | 0 | 1 | 0 | 3 | 3 |

## C.3 Coverage table – few objects

Table C.5: Coverage table of classical heuristics and learned heuristics with **sum readout** on planning tasks with **few objects**.

| | Eager | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Baseline | | | | Domain-dependent | | | | | | Domain-independent | | | | | | |
| Domain | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| Blocksworld (40) | 33 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 39 | **40** | 32 | 24 | **40** | 30 | 23 |
| Ferry (125) | 90 | **125** | 79 | **125** | **125** | **125** | **125** | **125** | **125** | 112 | 65 | **125** | 115 | 40 | **125** | 56 | 40 |
| Gripper (10) | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | 9 | **10** | **10** | **10** |
| Hanoi (8) | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | 6 |
| $n$-puzzle (20) | 10 | **20** | 10 | 19 | **20** | **20** | **20** | **20** | **20** | 5 | 1 | **20** | 17 | 12 | **20** | 4 | 1 |
| Sokoban (30) | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | 18 | 23 | 23 | **30** | **30** |
| Spanner (75) | 65 | 65 | 60 | 60 | 60 | 70 | **75** | 72 | 70 | **75** | **75** | 60 | 40 | 40 | 60 | 55 | 40 |
| VisitAll (40) | 10 | **40** | 10 | **40** | 39 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 10 | 13 | 31 | **40** | **40** |
| VisitSome (40) | 22 | **40** | 16 | 33 | 30 | 39 | **40** | **40** | **40** | 36 | 37 | 39 | 23 | 29 | 35 | 39 | 37 |

| | Lazy | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Baseline | | | | Domain-dependent | | | | | | Domain-independent | | | | | | |
| Domain | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| Blocksworld (40) | 33 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 37 | 31 | **40** | 32 | 23 |
| Ferry (125) | 88 | **125** | 82 | **125** | **125** | **125** | **125** | **125** | **125** | 118 | 82 | **125** | 119 | 59 | **125** | 59 | 41 |
| Gripper (10) | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** |
| Hanoi (8) | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | 7 |
| $n$-puzzle (20) | 10 | **20** | 10 | 19 | **20** | **20** | **20** | **20** | **20** | 10 | 8 | **20** | **20** | 11 | **20** | 10 | 5 |
| Sokoban (30) | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** |
| Spanner (75) | 65 | 65 | 69 | 70 | 70 | **75** | **75** | 74 | **75** | **75** | **75** | 60 | 56 | 60 | 65 | 55 | 40 |
| VisitAll (40) | 10 | **40** | 10 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 15 | 15 | 34 | **40** | **40** |
| VisitSome (40) | 22 | **40** | 16 | 35 | 33 | **40** | **40** | **40** | **40** | 34 | 34 | **40** | 25 | 32 | 37 | 38 | **40** |

Table C.6: Coverage table of classical heuristics and learned heuristics with **mean readout** on planning tasks with **few objects**.

| | Eager | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Baseline | | | | Domain-dependent | | | | | | Domain-independent | | | | | | |
| Domain | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| Blocksworld (40) | 33 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 33 | 34 |
| Ferry (125) | 90 | **125** | 79 | **125** | **125** | **125** | **125** | **125** | **125** | 116 | 107 | **125** | **125** | 113 | **125** | 97 | 78 |
| Gripper (10) | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | 8 | **10** | **10** | 9 | **10** |
| Hanoi (8) | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | 7 | 7 | **8** | 6 | **8** | 7 | 7 | 6 |
| $n$-puzzle (20) | 10 | **20** | 10 | 19 | **20** | **20** | **20** | **20** | **20** | 2 | 1 | 19 | 14 | **20** | **20** | 3 | 1 |
| Sokoban (30) | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | 27 | **30** | 17 | 15 | 23 | 15 | 27 |
| Spanner (75) | 65 | 65 | 60 | 60 | 60 | 70 | 70 | 70 | 69 | **75** | **75** | 58 | 52 | 70 | 61 | 55 | 50 |
| VisitAll (40) | 10 | **40** | 10 | **40** | 39 | **40** | **40** | **40** | **40** | **40** | **40** | 14 | 31 | 20 | 13 | **40** | **40** |
| VisitSome (40) | 22 | **40** | 16 | 33 | 30 | **40** | **40** | **40** | **40** | 38 | 37 | 19 | 29 | 38 | 29 | 39 | **40** |

| | Lazy | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Baseline | | | | Domain-dependent | | | | | | Domain-independent | | | | | | |
| Domain | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| Blocksworld (40) | 33 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 33 | 36 |
| Ferry (125) | 88 | **125** | 82 | **125** | **125** | **125** | **125** | **125** | **125** | 118 | 117 | **125** | **125** | 109 | **125** | 96 | 79 |
| Gripper (10) | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | 9 | **10** | **10** | **10** | **10** |
| Hanoi (8) | **8** | **8** | **8** | **8** | **8** | **8** | **8** | **8** | 7 | **8** | 7 | **8** | 7 | **8** | **8** | **8** | 6 |
| $n$-puzzle (20) | 10 | **20** | 10 | 19 | **20** | **20** | **20** | **20** | **20** | 10 | 5 | **20** | 19 | **20** | **20** | 8 | 3 |
| Sokoban (30) | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** | **30** |
| Spanner (75) | 65 | 65 | 69 | 70 | 70 | **75** | **75** | **75** | **75** | **75** | **75** | 60 | 58 | 70 | 70 | 55 | 50 |
| VisitAll (40) | 10 | **40** | 10 | **40** | **40** | **40** | **40** | **40** | **40** | **40** | **40** | 16 | 36 | 20 | 15 | **40** | **40** |
| VisitSome (40) | 22 | **40** | 16 | 35 | 33 | **40** | **40** | **40** | **40** | **40** | 39 | 23 | 35 | 39 | 34 | 39 | **40** |

## C.4 Coverage table – many objects

Table C.7: Coverage table of classical heuristics and learned heuristics with **sum readout** on planning tasks with **many objects**.

| | Eager | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Baseline | | | | Domain-dependent | | | | | | Domain-independent | | | | | | |
| Domain | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| Blocksworld (90) | - | 15 | - | 20 | 10 | 18 | **22** | 13 | 10 | 9 | 6 | 16 | - | 4 | 6 | - | - |
| Ferry (90) | - | **90** | - | 11 | 38 | 9 | 43 | 20 | 41 | - | - | 44 | - | 1 | 17 | - | - |
| Gripper (18) | 1 | **18** | - | **18** | 14 | 7 | 1 | 5 | 3 | 14 | 12 | 4 | - | - | - | 3 | 7 |
| Hanoi (18) | **1** | **1** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| $n$-puzzle (50) | - | **11** | - | - | **11** | 10 | 8 | - | 10 | - | - | 6 | - | - | 5 | - | - |
| Sokoban (90) | 42 | 81 | **82** | 52 | 45 | 30 | 31 | 29 | 34 | 27 | 27 | 20 | 12 | 3 | 18 | 18 | 18 |
| Spanner (90) | - | - | - | - | - | - | - | 1 | - | 15 | **50** | - | - | - | - | - | - |
| VisitAll (90) | - | **75** | - | 2 | 9 | 38 | 39 | 21 | 20 | 31 | 31 | 44 | 30 | - | 16 | 17 | 30 |
| VisitSome (90) | - | 7 | - | - | - | 2 | 8 | - | 2 | 4 | - | 3 | - | 2 | 2 | 7 | **12** |

| | Lazy | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Baseline | | | | Domain-dependent | | | | | | Domain-independent | | | | | | |
| Domain | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| Blocksworld (90) | - | 13 | - | 21 | 10 | 17 | **23** | 14 | 10 | 8 | 7 | 15 | - | 7 | 8 | - | - |
| Ferry (90) | - | **90** | - | 16 | 69 | 11 | 40 | 19 | 40 | - | - | 44 | - | - | 13 | - | - |
| Gripper (18) | 1 | **18** | - | **18** | **18** | 10 | 12 | 17 | **18** | 15 | 12 | 5 | 1 | - | 13 | 7 | 9 |
| Hanoi (18) | **1** | **1** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| $n$-puzzle (50) | - | 11 | - | - | **14** | 12 | 12 | 5 | 13 | - | - | 7 | - | - | 6 | - | - |
| Sokoban (90) | 42 | 81 | **90** | 63 | 72 | 32 | 34 | 29 | 34 | 33 | 27 | 39 | 18 | 18 | 18 | 25 | 24 |
| Spanner (90) | - | - | - | - | - | 10 | 3 | 10 | 1 | 45 | **90** | - | - | - | - | - | - |
| VisitAll (90) | - | **73** | - | 4 | 14 | 40 | 30 | 24 | 17 | 30 | 25 | 40 | 28 | - | 17 | 15 | 25 |
| VisitSome (90) | - | 5 | - | - | - | - | **13** | 3 | 4 | 5 | - | 7 | 1 | 2 | 5 | 7 | 12 |

Table C.8: Coverage table of classical heuristics and learned heuristics with **mean readout** on planning tasks with **many objects**.

| | Eager | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Baseline | | | | Domain-dependent | | | | | | Domain-independent | | | | | | |
| Domain | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| Blocksworld (90) | - | 15 | - | 20 | 10 | 25 | 26 | **32** | 31 | 19 | 8 | 6 | 4 | 9 | 10 | - | - |
| Ferry (90) | - | **90** | - | 11 | 38 | 35 | 40 | 39 | 40 | - | - | 14 | 5 | 30 | 12 | - | - |
| Gripper (18) | 1 | **18** | - | **18** | 14 | 7 | 7 | 5 | 7 | 7 | 6 | 2 | - | 4 | 2 | 3 | 5 |
| Hanoi (18) | **1** | **1** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| $n$-puzzle (50) | - | **11** | - | - | **11** | 5 | 8 | 8 | 6 | - | - | 6 | - | 8 | 5 | - | - |
| Sokoban (90) | 42 | 81 | **82** | 52 | 45 | 38 | 31 | 39 | 38 | 28 | 27 | 9 | 4 | 30 | 3 | 36 | 36 |
| Spanner (90) | - | - | - | - | - | - | - | - | - | **15** | 9 | - | - | - | - | - | - |
| VisitAll (90) | - | **75** | - | 2 | 9 | 15 | 15 | 10 | 5 | 10 | 10 | - | - | - | - | 12 | 12 |
| VisitSome (90) | - | 7 | - | - | - | 7 | 6 | - | 1 | 6 | 3 | - | - | - | - | **10** | 9 |

| | Lazy | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Baseline | | | | Domain-dependent | | | | | | Domain-independent | | | | | | |
| Domain | blind | $h^{gc}$ | $h^{max}$ | $h^{add}$ | $h^{FF}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ | $h^{SDG}$ | $h^{SDG^E}$ | $h^{FDG}$ | $h^{FDG^E}$ | $h^{LDG}$ | $h^{LDG^E}$ |
| Blocksworld (90) | - | 13 | - | 21 | 10 | 26 | 32 | **33** | 32 | 20 | 7 | 5 | 4 | 10 | 9 | - | - |
| Ferry (90) | - | **90** | - | 16 | 69 | 33 | 40 | 40 | 40 | - | - | 13 | 5 | 28 | 12 | - | - |
| Gripper (18) | 1 | **18** | - | **18** | **18** | 7 | 8 | 5 | 7 | 7 | 6 | 2 | - | 4 | 1 | 6 | 8 |
| Hanoi (18) | **1** | **1** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| $n$-puzzle (50) | - | 11 | - | - | **14** | 8 | 8 | 9 | 9 | - | - | 3 | - | 8 | 7 | - | - |
| Sokoban (90) | 42 | 81 | **90** | 63 | 72 | 35 | 39 | 45 | 35 | 31 | 31 | 20 | 18 | 37 | 18 | 36 | 36 |
| Spanner (90) | - | - | - | - | - | 1 | - | 2 | 1 | **25** | 9 | - | - | - | - | - | - |
| VisitAll (90) | - | **73** | - | 4 | 14 | 15 | 15 | 10 | 5 | 10 | 10 | - | - | - | - | 15 | 15 |
| VisitSome (90) | - | 5 | - | - | - | 7 | 4 | - | 1 | 3 | 3 | - | - | - | - | **9** | 7 |

## C.5 Coverage plots of runtime and plan quality

### C.5.1 Blocksworld

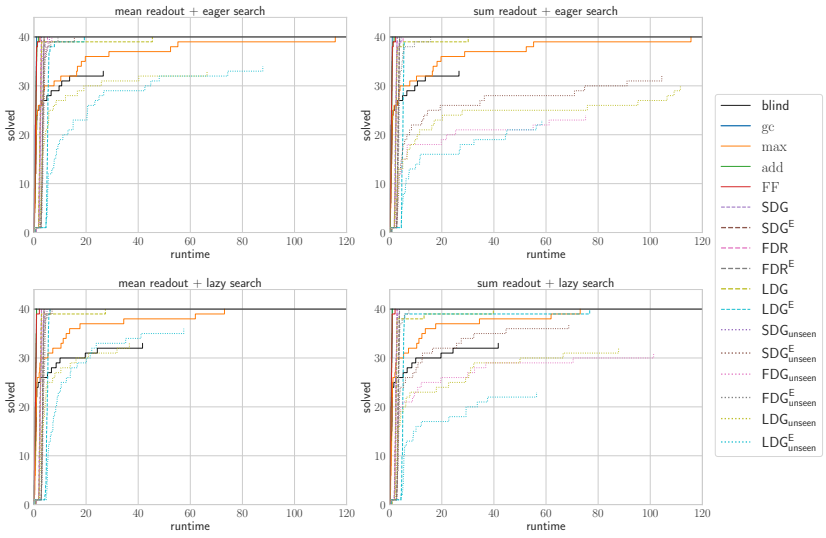



Figure C.1: Cumulative coverage over runtime on seen/small size Blocksworld instances. Total number of problems: 40.

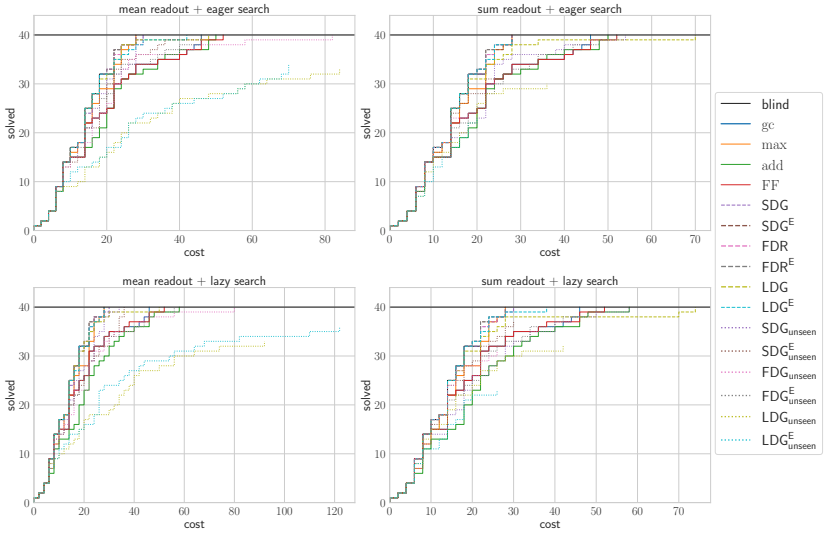



Figure C.2: Cumulative coverage over plan cost on seen/small size Blocksworld instances. Total number of problems: 40.

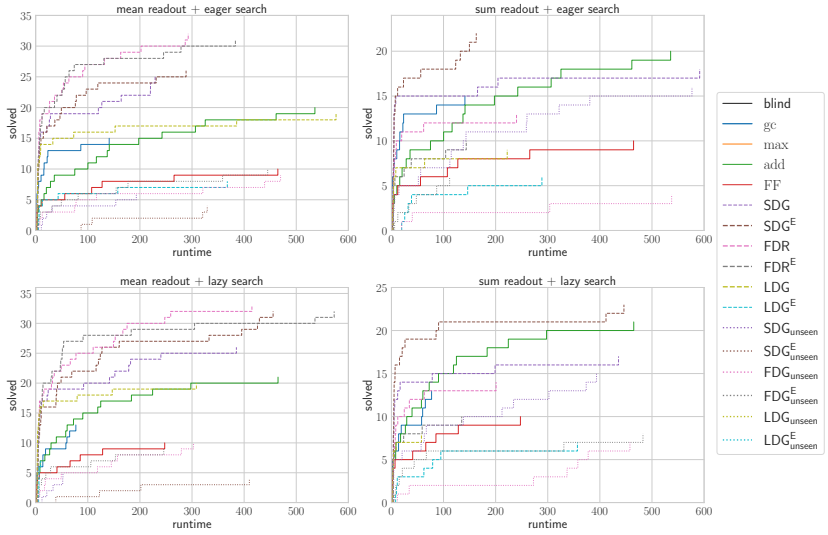



Figure C.3: Cumulative coverage over runtime on unseen/large size BLOCKSWORLD instances. Total number of problems: 90.

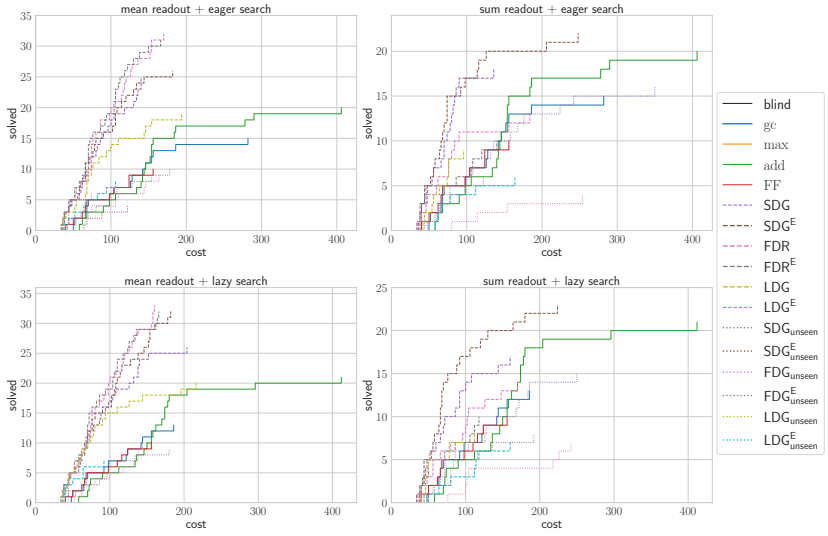



Figure C.4: Cumulative coverage over plan cost on unseen/large size BLOCKSWORLD instances. Total number of problems: 90.

### C.5.2 Ferry

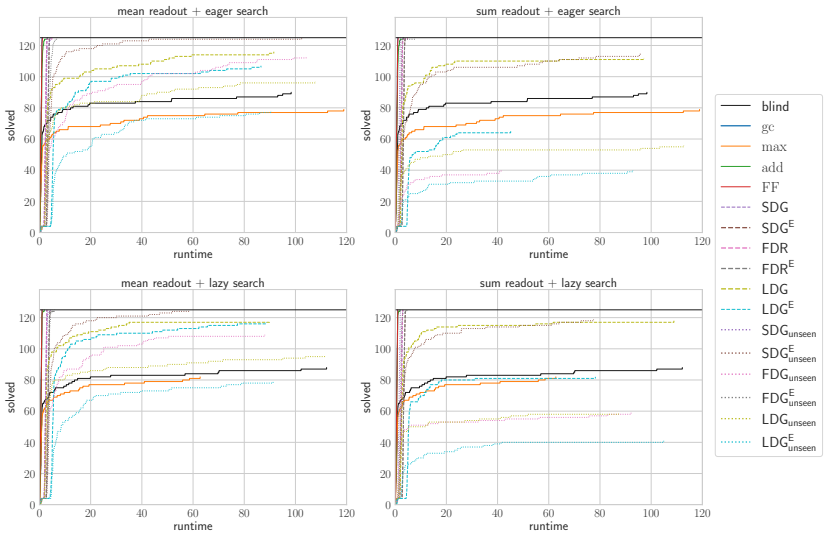



Figure C.5: Cumulative coverage over runtime on seen/small size Ferry instances. Total number of problems: 125.

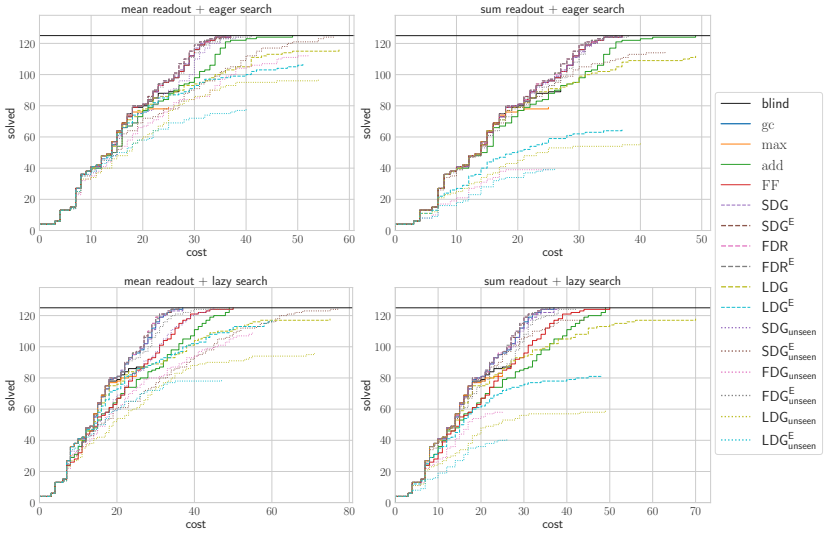



Figure C.6: Cumulative coverage over plan cost on seen/small size Ferry instances. Total number of problems: 125.

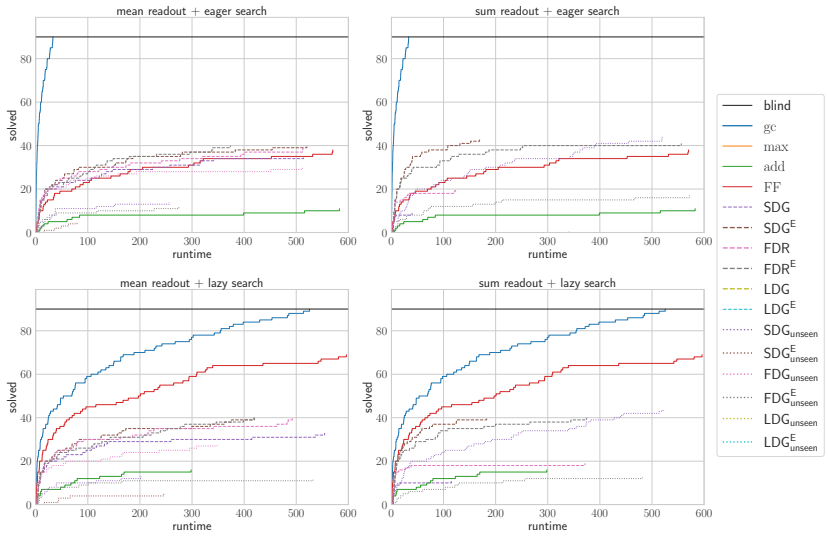



Figure C.7: Cumulative coverage over runtime on unseen/large size Ferry instances. Total number of problems: 90.

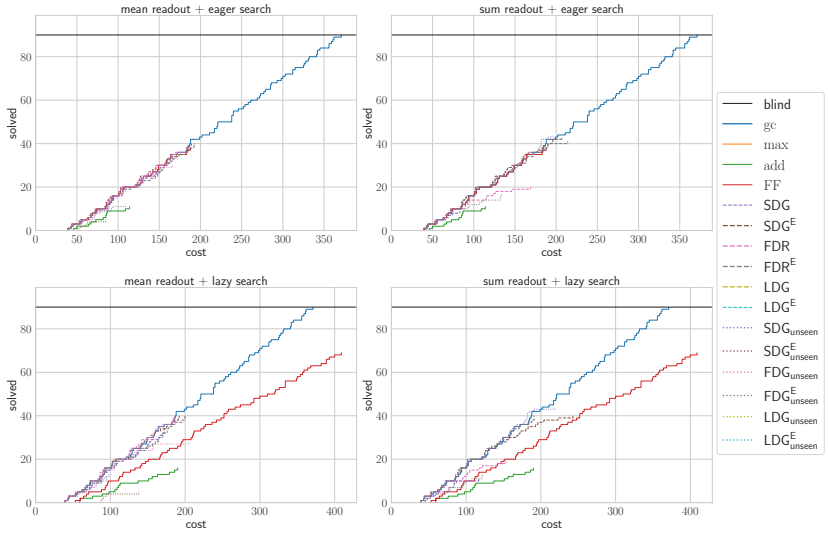



Figure C.8: Cumulative coverage over plan cost on unseen/large size Ferry instances. Total number of problems: 90.

### C.5.3 Gripper

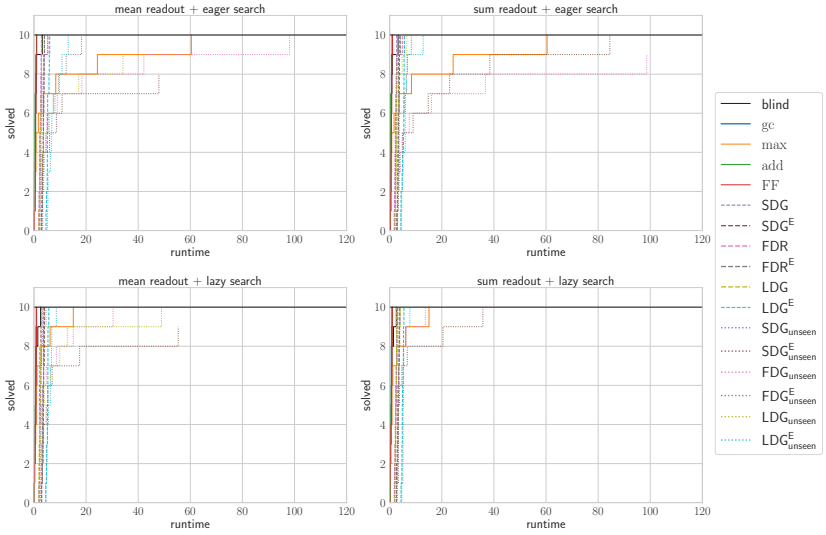



Figure C.9: Cumulative coverage over runtime on seen/small size Gripper instances. Total number of problems: 10.

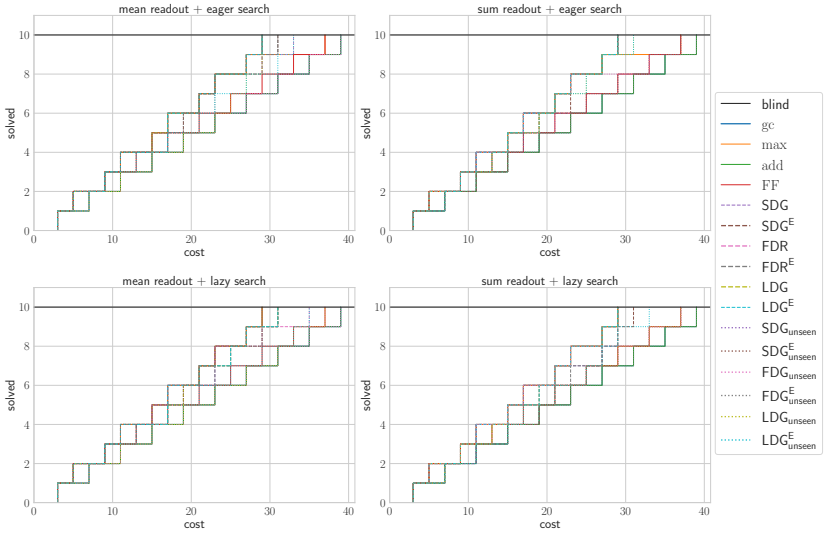



Figure C.10: Cumulative coverage over plan cost on seen/small size Gripper instances. Total number of problems: 10.

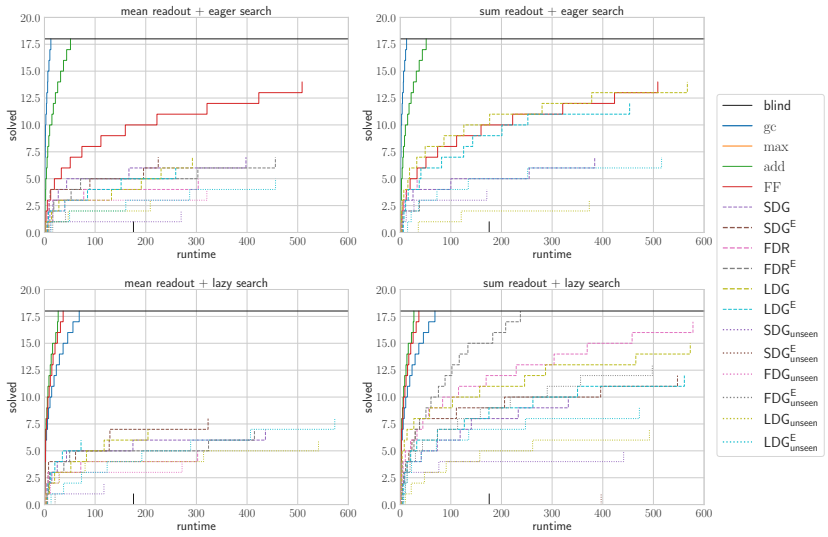



Figure C.11: Cumulative coverage over runtime on unseen/large size GRIPPER instances. Total number of problems: 18.

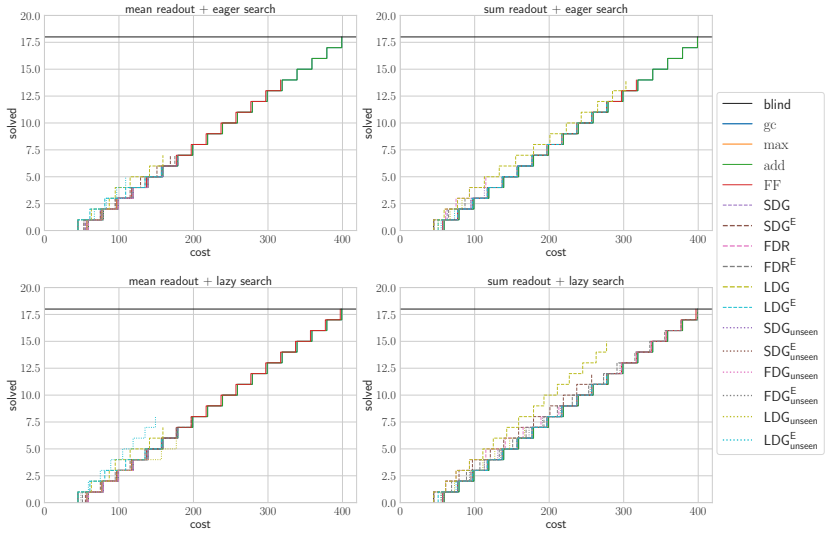



Figure C.12: Cumulative coverage over plan cost on unseen/large size GRIPPER instances. Total number of problems: 18.

### C.5.4 Hanoi

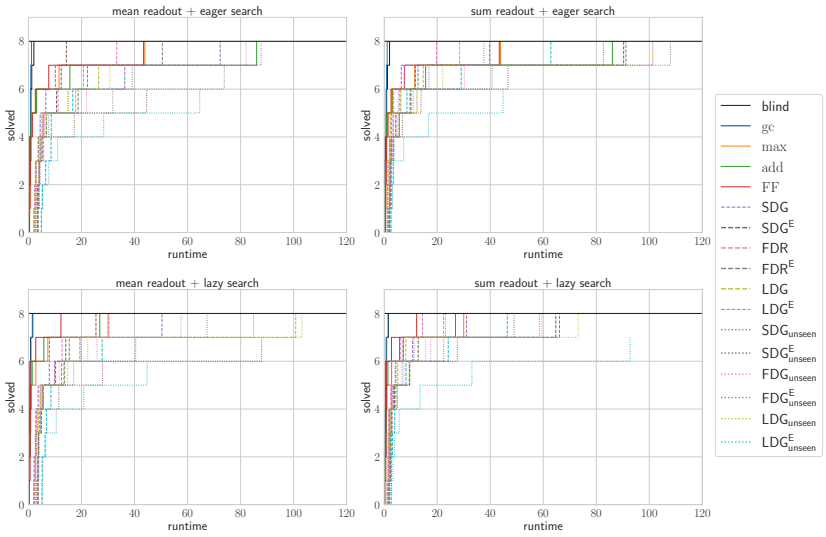



Figure C.13: Cumulative coverage over runtime on seen/small size HANOI instances. Total number of problems: 8.

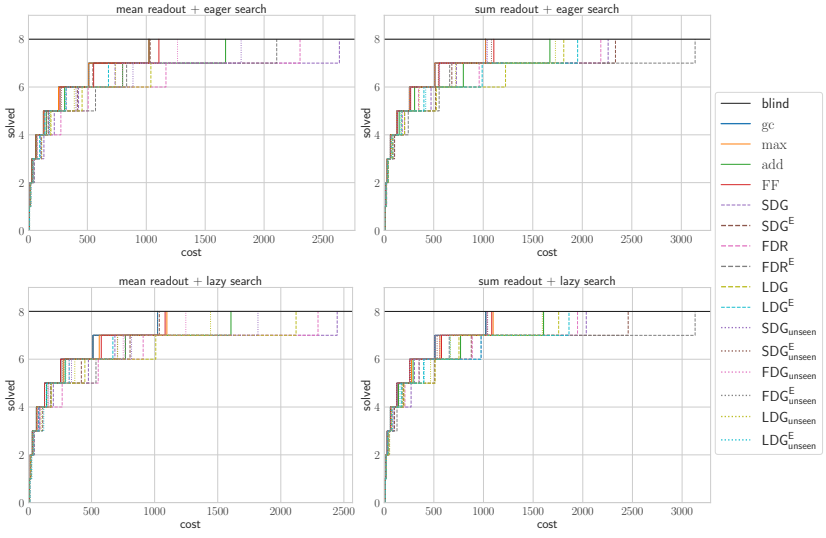



Figure C.14: Cumulative coverage over plan cost on seen/small size HANOI instances. Total number of problems: 8.

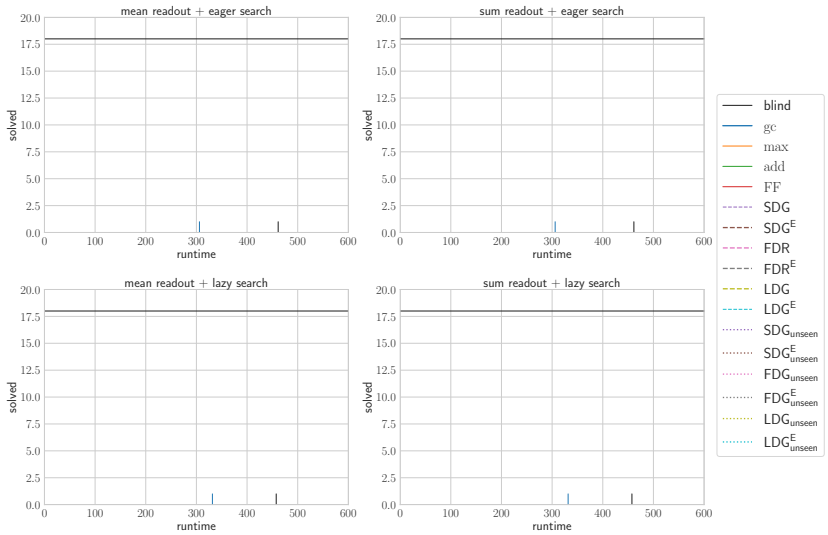



Figure C.15: Cumulative coverage over runtime on unseen/large size HANOI instances. Total number of problems: 18.

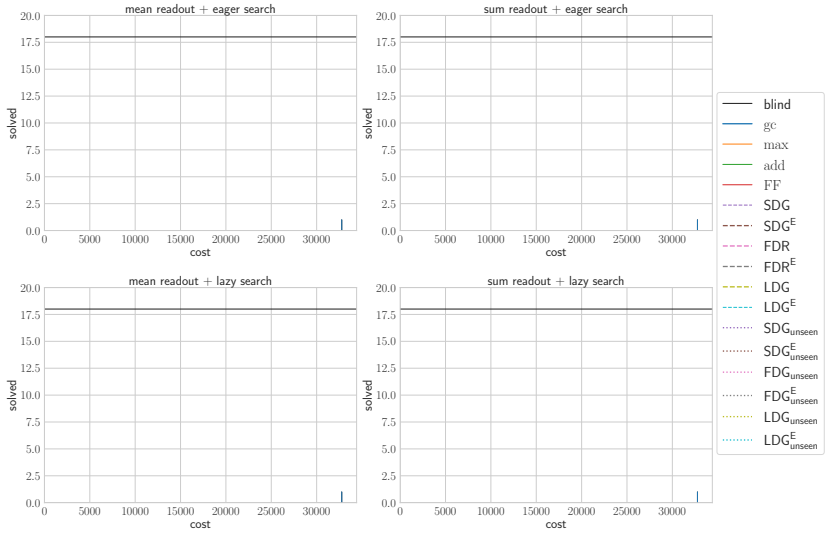



Figure C.16: Cumulative coverage over plan cost on unseen/large size HANOI instances. Total number of problems: 18.

### C.5.5 $n$-puzzle

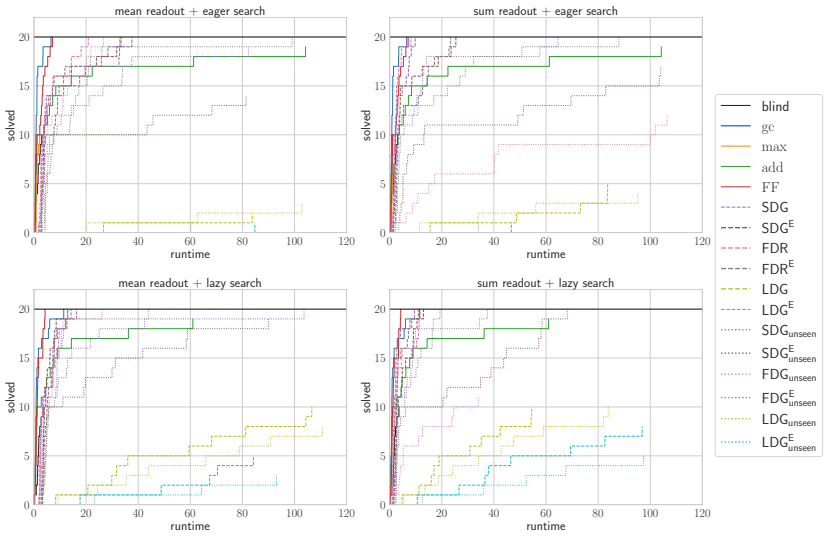



Figure C.17: Cumulative coverage over runtime on seen/small size $n$-PUZZLE instances. Total number of problems: 20.

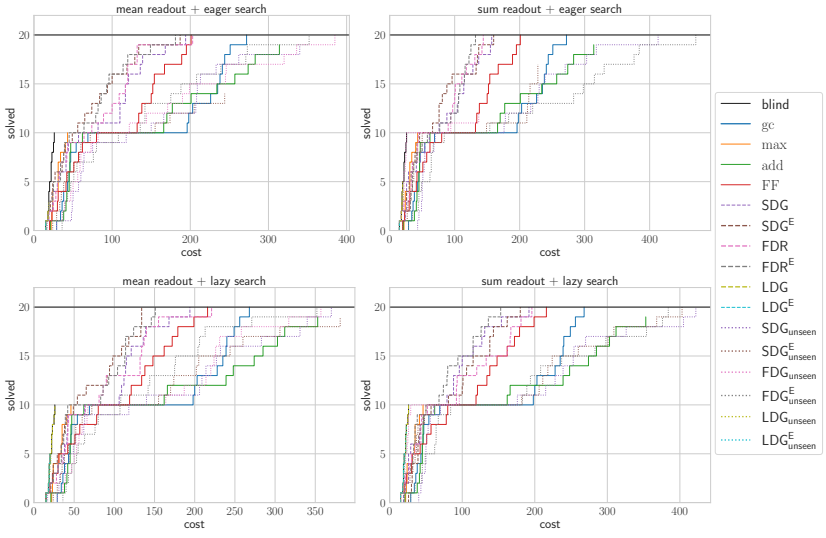



Figure C.18: Cumulative coverage over plan cost on seen/small size $n$-PUZZLE instances. Total number of problems: 20.

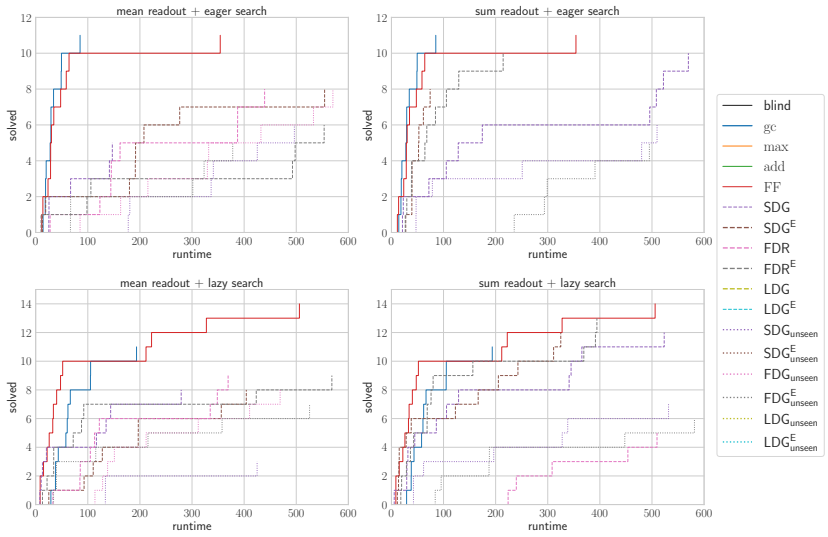



Figure C.19: Cumulative coverage over runtime on unseen/large size $n$-PUZZLE instances. Total number of problems: 50.

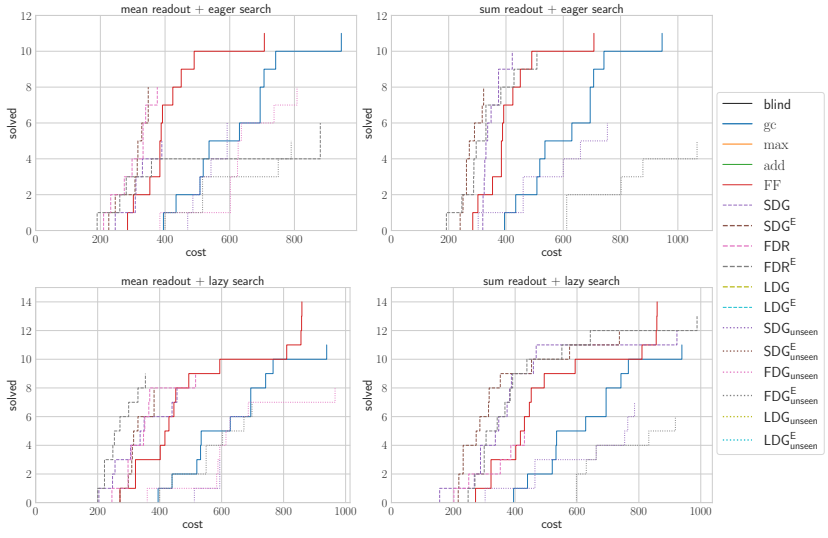



Figure C.20: Cumulative coverage over plan cost on unseen/large size $n$-PUZZLE instances. Total number of problems: 50.

### C.5.6 Sokoban

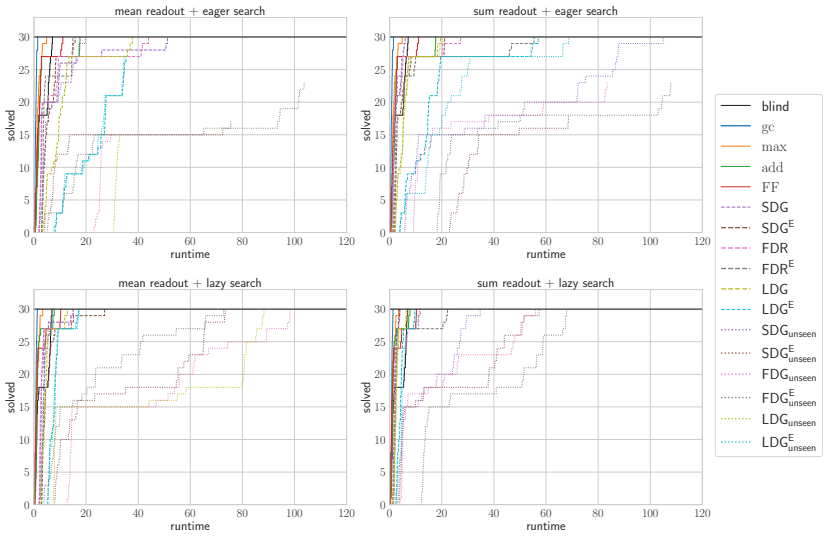



Figure C.21: Cumulative coverage over runtime on seen/small size Sokoban instances. Total number of problems: 30.

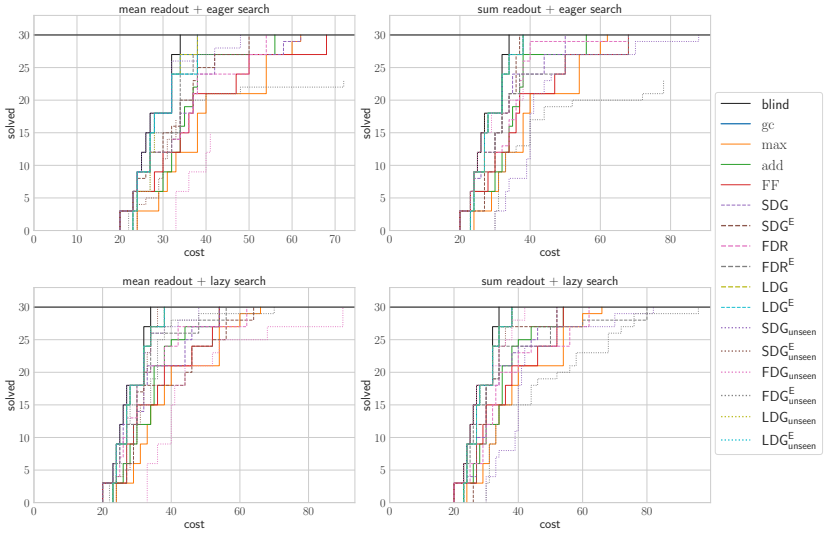



Figure C.22: Cumulative coverage over plan cost on seen/small size Sokoban instances. Total number of problems: 30.

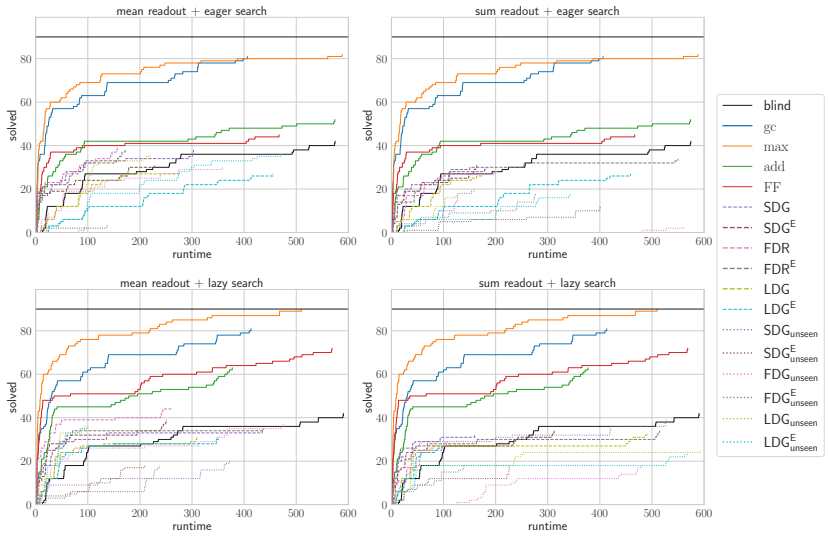



Figure C.23: Cumulative coverage over runtime on unseen/large size SOKOBAN instances. Total number of problems: 90.

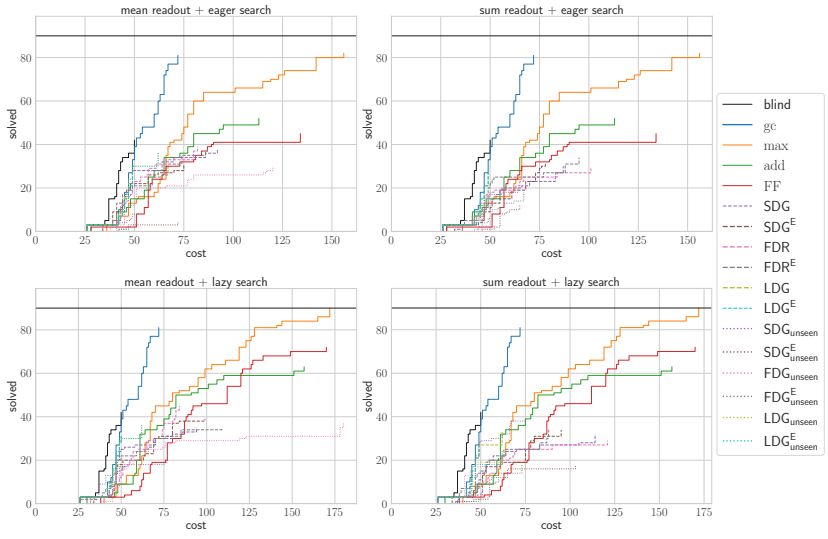



Figure C.24: Cumulative coverage over plan cost on unseen/large size SOKOBAN instances. Total number of problems: 90.

### C.5.7 Spanner

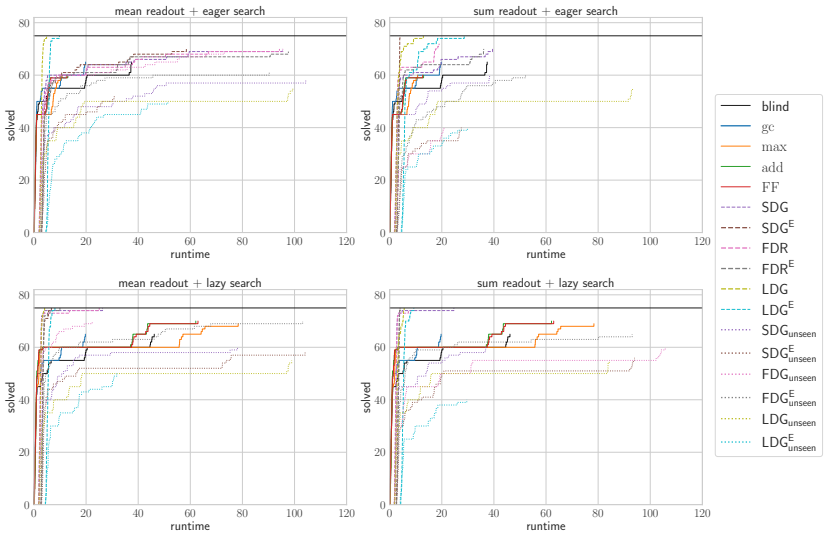



Figure C.25: Cumulative coverage over runtime on seen/small size Spanner instances. Total number of problems: 75.

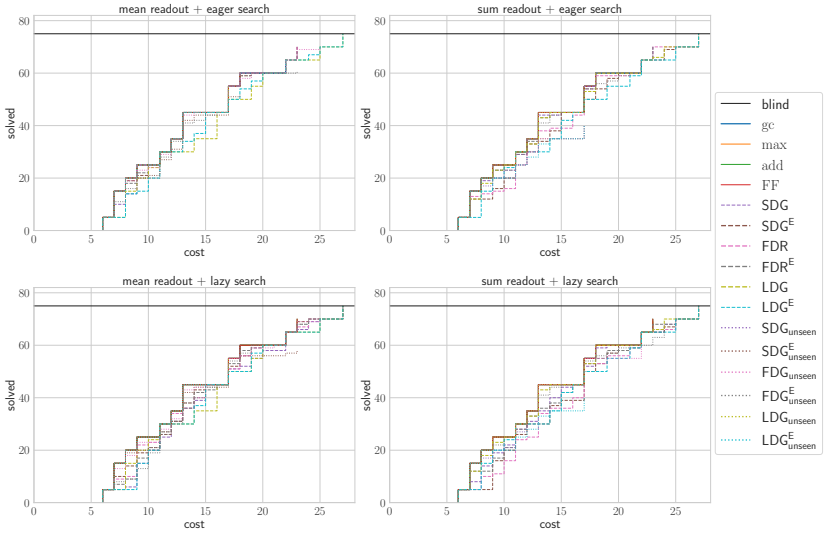



Figure C.26: Cumulative coverage over plan cost on seen/small size Spanner instances. Total number of problems: 75.

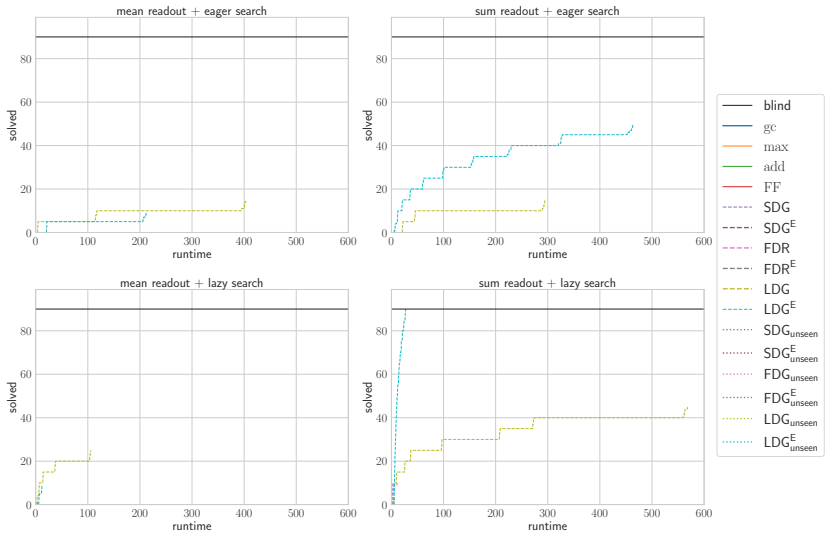



Figure C.27: Cumulative coverage over runtime on unseen/large size SPANNER instances. Total number of problems: 90.

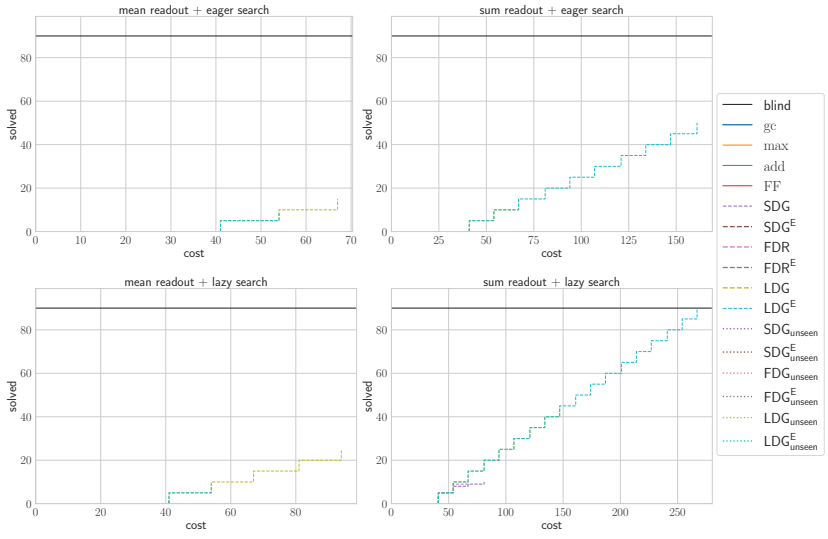



Figure C.28: Cumulative coverage over plan cost on unseen/large size SPANNER instances. Total number of problems: 90.

### C.5.8 VisitAll

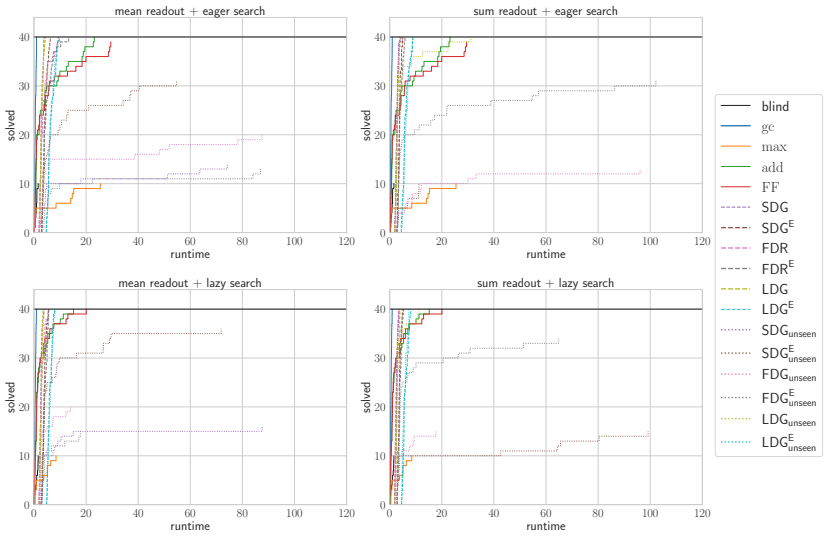



Figure C.29: Cumulative coverage over runtime on seen/small size VisitAll instances. Total number of problems: 40.

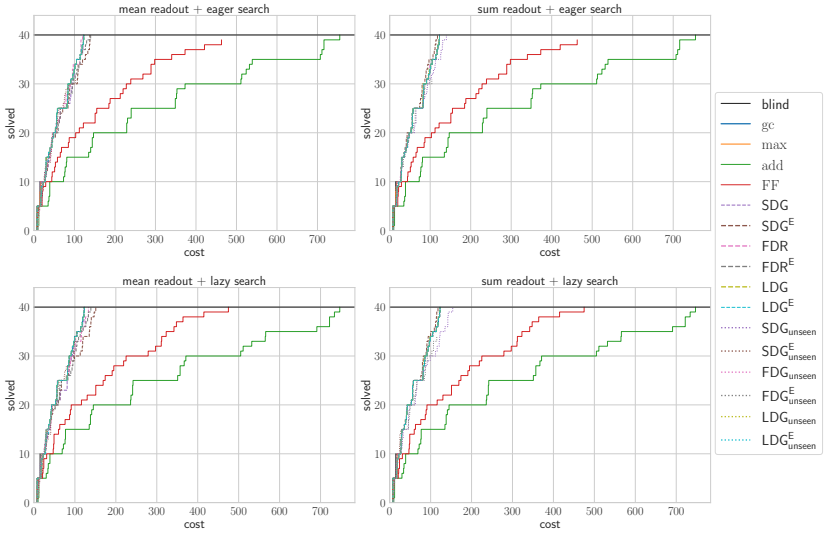



Figure C.30: Cumulative coverage over plan cost on seen/small size VisitAll instances. Total number of problems: 40.

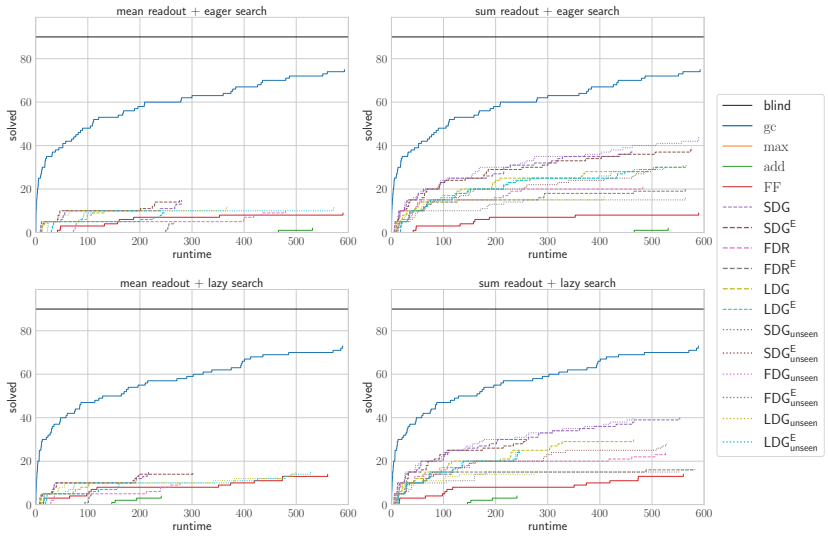



Figure C.31: Cumulative coverage over runtime on unseen/large size VisitAll instances. Total number of problems: 90.

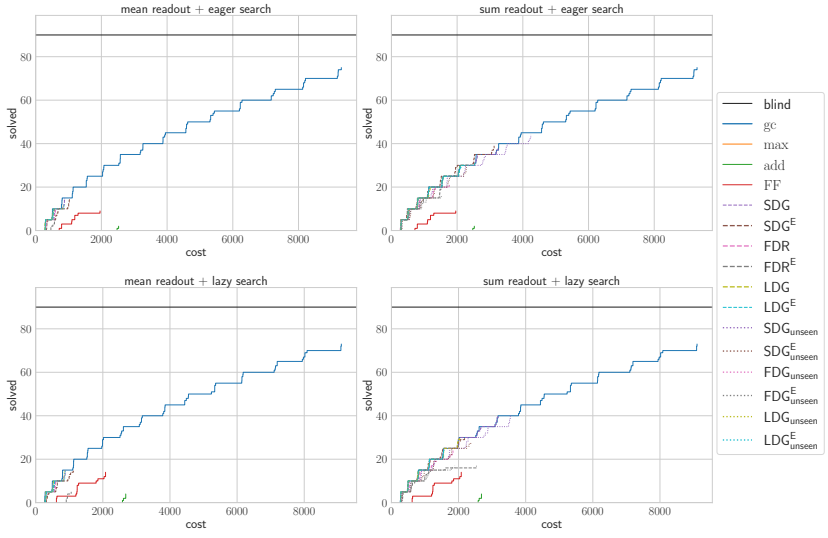



Figure C.32: Cumulative coverage over plan cost on unseen/large size VisitAll instances. Total number of problems: 90.

### C.5.9 VisitSome

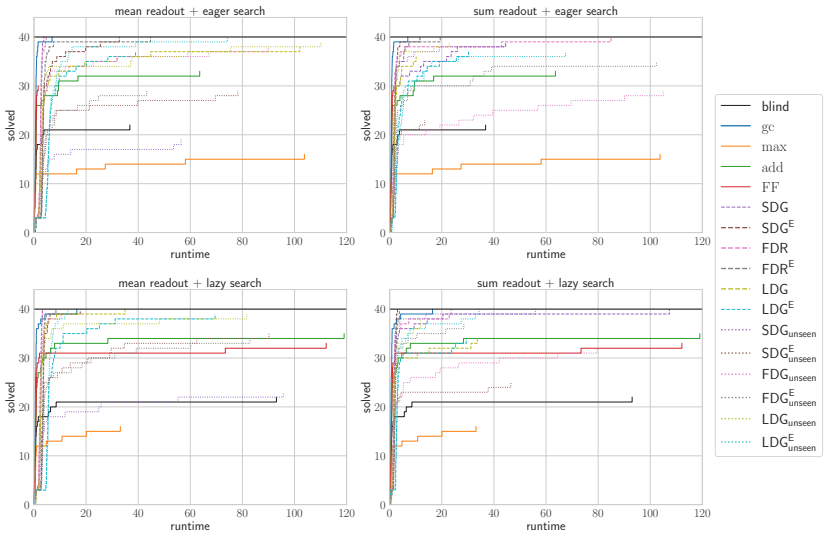



Figure C.33: Cumulative coverage over runtime on seen/small size VisitSome instances. Total number of problems: 40.

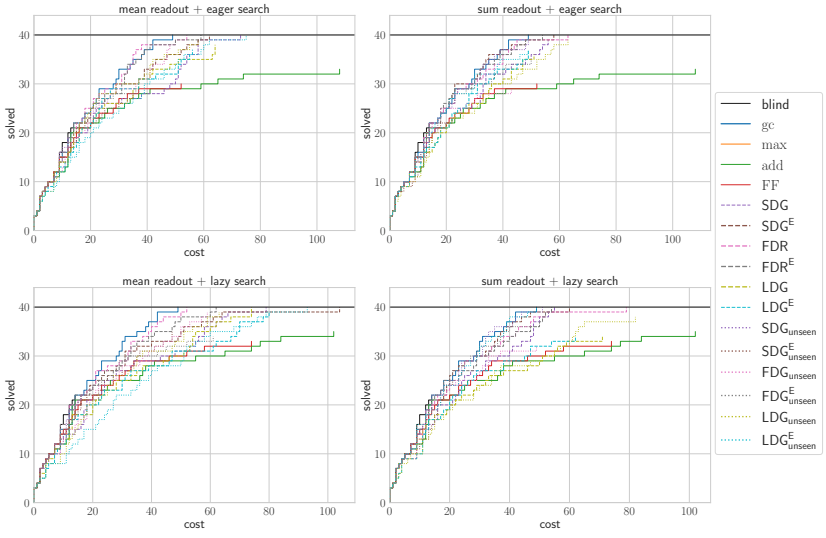



Figure C.34: Cumulative coverage over plan cost on seen/small size VisitSome instances. Total number of problems: 40.

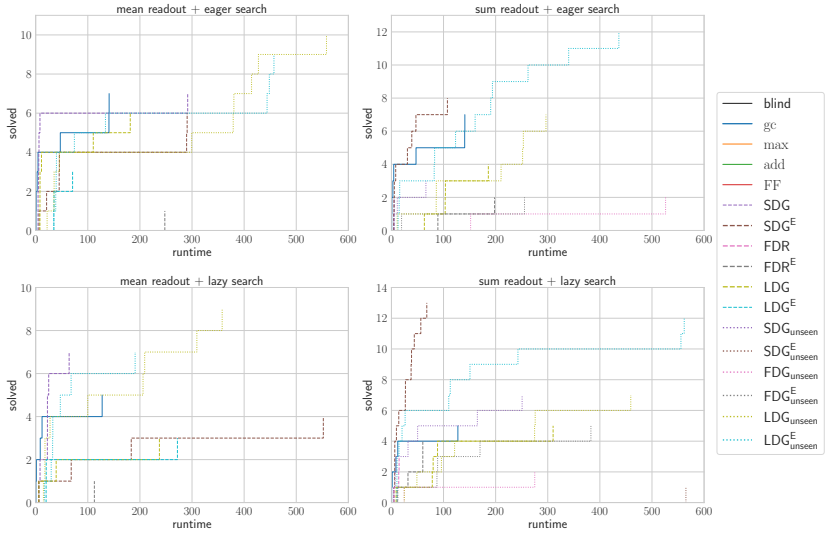



Figure C.35: Cumulative coverage over runtime on unseen/large size VisitSome instances. Total number of problems: 90.

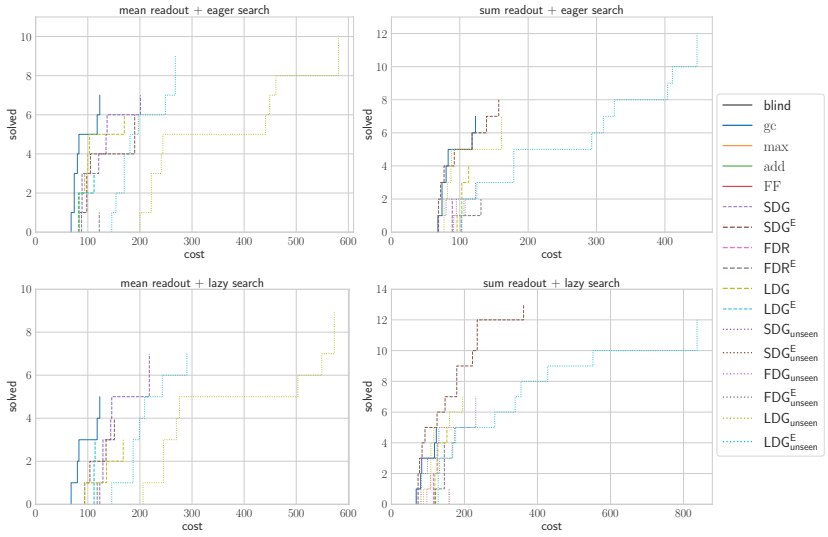



Figure C.36: Cumulative coverage over plan cost on unseen/large size VisitSome instances. Total number of problems: 90.

# Bibliography


Ralph Abboud, İsmail İlkan Ceylan, Martin Grohe, and Thomas Lukasiewicz. The surprising power of graph neural networks with random node initialization. *CoRR*, abs/2010.01179, 2020.

Ralph Abboud, İsmail İlkan Ceylan, Martin Grohe, and Thomas Lukasiewicz. The surprising power of graph neural networks with random node initialization. In *Proc. of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

Mohammad Abdulaziz, Michael Norrish, and Charles Gretton. Exploiting symmetries by planning for a descriptive quotient. In *Proc. of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. *Nat. Mach. Intell.*, 1(8): 356–363, 2019.

Vidal Alcázar and Álvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In *Proc. of the 25th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 2–6, 2015.

Ron Alford, Pascal Bercher, and David Aha. Tight bounds for HTN planning with task insertion. In *Proc. of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1502–1508, 2015.

Dario Amodei and Danny Hernandez. Ai and compute, 2018. Accessed from https://openai.com/research/ai-and-compute on May 8, 2023.

Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Métivier, and Sylvie Pesty. A review of learning planning action models. *Knowl. Eng. Rev.*, 33:e20, 2018.

Christer Bäckström and Inger Klein. Planning in polynomial time: the SAS-PUBS class. *Comput. Intell.*, 7:181–197, 1991.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Comput. Intell.*, 11:625–656, 1995.

Aniket (Nick) Bajpai, Sankalp Garg, and Mausam. Transfer of deep reactive policies for MDP planning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 10988–10998, 2018.

Pablo Barceló, Egor V Kostylev, Mikael Monet, Jorge Pérez, Juan Reutter, and Juan-Pablo Silva. The logical expressiveness of graph neural networks. In *8th International Conference on Learning Representations (ICLR)*, 2020.

Pablo Barceló, Floris Geerts, Juan Reutter, and Maksimilian Ryschkov. Graph neural networks with local graph parameters. *Neural Information Processing Systems (NeurIPS)*, 34, 2021.

Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

Dimitri P Bertsekas and John N Tsitsiklis. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16(3):580–595, 1991.

Christopher Bishop. Mixture density networks. Technical Report NCRG/94/004, 1994.

Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997.

Blai Bonet and Hector Geffner. Labeled RTDP: improving the convergence of real-time dynamic programming. In *Proc. 13th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 12–21, 2003.

Blai Bonet and Hector Geffner. Learning first-order symbolic representations for planning from the structure of the state space. In *Proc. of the 24th European Conference on Artificial Intelligence (ECAI)*, volume 325, pages 2322–2329, 2020.

Blai Bonnet and Héctor Geffner. HSP: Heuristic search planner. *First description of HSP at the AIPS-98 Planning Competition*, 1998.

Giorgos Bouritsas, Fabrizio Frasca, Stefanos P Zafeiriou, and Michael Bronstein. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

Zhaoxing Bu, Roni Stern, Ariel Felner, and Robert Craig Holte. A* with lookahead re-evaluated. In *Proc. of the 7th Annual Symposium on Combinatorial Search (SOCS)*, 2014.

Ethan Burns, Sofia Lemons, Wheeler Ruml, and Rong Zhou. Best-first heuristic search for multicore machines. *J. Artif. Intell. Res.*, 39:689–743, 2010.

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

Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Velickovic. Combinatorial optimization and reasoning with graph neural networks. In *Proc. of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4348–4355, 2021.

Dillon Chen and Pascal Bercher. Fully observable nondeterministic htn planning – formalisation and complexity results. In *ICAPS 2021*, pages 74–84, 2021.

Dillon Chen and Pascal Bercher. Flexible FOND HTN planning: A complexity analysis. In *Proc. of the 32nd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 26–34, 2022.

Dillon Chen, Felipe Trevizan, and Sylvie Thiébaux. Heuristic Search for Multi-Objective Probabilistic Planning. In *Proc. of 37th AAAI Conference on Artificial Intelligence*, 2023.

Stephen V. Chenoweth. On the np-hardness of blocks world. In *Proc. of the 9th National Conference on Artificial Intelligence*, pages 623–628, 1991.

Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Francès. Lifted successor generation using query optimization techniques. In *Proc. of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 80–89, 2020.

Augusto B. Corrêa, Guillem Francès, Florian Pommerening, and Malte Helmert. Delete-relaxation heuristics for lifted classical planning. In *Proc. of the 31st International Conference on Automated Planning and Scheduling (ICAPS)*, pages 94–102, 2021.

Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Francès. The FF heuristic for lifted classical planning. In *Proc. of the 36th AAAI Conference on Artificial Intelligence*, pages 9716–9723, 2022.

Joseph Culberson. Sokoban is pspace-complete. In *Proc. of the International Conference on Fun with Algorithms*, pages 65–76, 1997.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3837–3845, 2016.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. In *7th International Conference on Learning Representations (ICLR)*, 2019.

Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *CoRR*, abs/2012.09699, 2020.

Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

Vijay Prakash Dwivedi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Graph neural networks with learnable structural and positional representations. *International Conference on Machine Learning (ICLR)*, 2022.

Stefan Edelkamp. Planning with pattern databases. In *Proc. of the 6th European Conference on Planning (ECP)*, pages 13–24, 2001.

Marco Ernandes and Marco Gori. Likely-admissible and sub-symbolic heuristics. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proc. of the 16th European Conference on Artificial Intelligence (ECAI)*, pages 613–617, 2004.

Kutluhan Erol, Dana S. Nau, and V.S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artif. Intell.*, 76(1):75–88, 1995.

Kutluhan Erol, James Hendler, and Dana S Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996.

Matthew P. Evett, James A. Hendler, Ambuj Mahanti, and Dana S. Nau. PRA*: Massively parallel heuristic search. *J. Parallel Distributed Comput.*, 25(2):133–143, 1995.

Ariel Felner, Sarit Kraus, and Richard E. Korf. KBFS: k-best-first search. *Ann. Math. Artif. Intell.*, 39(1-2):19–39, 2003.

Patrick Ferber, Malte Helmert, and Jörg Hoffmann. Neural network heuristics for classical planning: A study of hyperparameter space. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *Proc. of the 24th European Conference on Artificial Intelligence (ECAI)*, volume 325, pages 2346–2353, 2020.

Patrick Ferber, Florian Geißer, Felipe W. Trevizan, Malte Helmert, and Jörg Hoffmann. Neural network heuristic functions for classical planning: Bootstrapping and comparison to other methods. In Akshat Kumar, Sylvie Thiébaux, Pradeep Varakantham, and William Yeoh, editors, *Proc. of 32nd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 583–587, 2022.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.

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

Guillem Francès, Augusto B. Corrêa, Cedric Geissmann, and Florian Pommerening. Generalized potential heuristics for classical planning. In *Proc. of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5554–5561, 2019.

M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

Sankalp Garg, Aniket Bajpai, and Mausam. Size independent neural transfer for RDDL planning. In J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava, editors, *Proc. of 29th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 631–636, 2019.

Sankalp Garg, Aniket Bajpai, and Mausam. Symbolic network: Generalized neural policies for relational mdps. In *International Conference on Machine Learning (ICML)*, volume 119, pages 3397–3407, 2020.

Caelan Reed Garrett, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning to rank for synthesizing planning heuristics. In *Proc. of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3089–3095, 2016.

Hector Geffner. Model-free, model-based, and general intelligence. In *Proc. of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 10–17, 2018.

Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013.

Clement Gehring, Masataro Asai, Rohan Chitnis, Tom Silver, Leslie Pack Kaelbling, Shirin Sohrabi, and Michael Katz. Reinforcement learning for classical planning: Viewing heuristics as dense reward generators. In *Proc. of the 32nd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 588–596, 2022.

F. Geißer, P. Haslum, Sylvie Thiébaux, and Felipe Trevizan. Admissible heuristics for multi-objective planning. In *Proc. 32nd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 100–109, 2022.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning (ICML)*, pages 1263–1272, 2017.

Daniel Gnad, Álvaro Torralba, Martín Ariel Domínguez, Carlos Areces, and Facundo Bustos. Learning how to ground a plan - partial grounding in classical planning. In *Proc. of the 23rd AAAI Conference on Artificial Intelligence*, pages 7602–7609, 2019.

Pawel Gomoluch, Dalal Alrajeh, Alessandra Russo, and Antonio Bucchiarone. Towards learning domain-independent planning heuristics. *CoRR*, abs/1707.06895, 2017.

Edward Groshev, Maxwell Goldstein, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks. In *Proc. of the 28th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 408–416, 2018.

Naresh Gupta and Dana S. Nau. Complexity results for blocks-world planning. In *Proc. of the 9th National Conference on Artificial Intelligence*, pages 629–633, 1991.

Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artif. Intell.*, 56(2-3):223–254, 1992.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1024–1034, 2017.

William L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159, 2020.

Eric A. Hansen and Shlomo Zilberstein. LAO$^*$: A heuristic search algorithm that finds solutions with loops. *Artif. Intell.*, 129(1-2):35–62, 2001.

Patrik Haslum. Reducing accidental complexity in planning problems. In *Proc. of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1898–1903, 2007.

Patrik Haslum. $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to $h^m$. In *Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.

Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. of the 22nd AAAI Conference on Artificial Intelligence*, pages 1007–1012, 2007.

Patrik Haslum, John K. Slaney, and Sylvie Thiébaux. Minimal landmarks for optimal delete-free planning. In *Proc. of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 2012.

Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

Xin He, Yapeng Yao, Zhiwen Chen, Jianhua Sun, and Hao Chen. Efficient parallel a* search on multi-gpu system. *Future Gener. Comput. Syst.*, 123:35–47, 2021.

Robert A. Hearn and Erik D. Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theor. Comput. Sci.*, 343(1-2):72–96, 2005.

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

Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6):503–535, 2009.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 162–169, 2009.

Malte Helmert and Hector Geffner. Unifying the causal graph and additive heuristics. In *Proc. of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 140–147, 2008.

Malte Helmert and Gabriele Röger. How good is almost perfect? In *Proc. of the 23rd AAAI Conference on Artificial Intelligence*, pages 944–949, 2008.

Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proc. of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 176–183, 2007.

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

Daniel Höller and Gregor Behnke. Encoding lifted classical planning in propositional logic. In *Proc. of the 32nd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 134–144, 2022.

Robert C. Holte. Common misconceptions concerning heuristic search. In *Proc. of the 3rd Annual Symposium on Combinatorial Search (SOCS)*, 2010.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Neural Information Processing Systems (NeurIPS)*, 2020.

León Illanes, Xi Yan, Rodrigo Toro Icarte, and Sheila McIlraith. Symbolic plans as high-level instructions for reinforcement learning. In *Proc. of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 540–550, 2020.

John J Irwin, Teague Sterling, Michael M Mysinger, Erin S Bolstad, and Ryan G Coleman. Zinc: a free tool to discover chemistry for biology. *Journal of chemical information and modeling*, 52(7):1757–1768, 2012.

Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Learning heuristic functions for large state spaces. *Artif. Intell.*, 175(16):2075–2098, 2011.

Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

Brendan Juba and Roni Stern. Learning probably approximately complete and safe action models for stochastic worlds. In *Proc. of the 36th AAAI Conference on Artificial Intelligence*, pages 9795–9804, 2022.

Brendan Juba, Hai S. Le, and Roni Stern. Safe learning of lifted action models. In *Proc. of the 18th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 379–389, 2021.

Rushang Karia and Siddharth Srivastava. Learning generalized relational heuristic networks for model-agnostic planning. In *Proc. of the 35th AAAI Conference on Artificial Intelligence*, pages 8064–8073, 2021.

Michael Katz and Carmel Domshlak. Optimal additive composition of abstraction-based admissible heuristics. In *Proc. of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 174–181, 2008.

Michael Katz, Shirin Sohrabi, Horst Samulowitz, and Silvan Sievers. Delfi: Online planner selection for cost-optimal planning. *Working Notes of the 9th International Planning Competition*, pages 57–64, 2018.

Henry Kautz and Bart Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. *Working notes of the AIPS-98 Workshop on Planning as Combinatorial Search*, 1998.

Henry Kautz, Bart Selman, and Jörg Hoffmann. SatPlan: Planning as Satisfiability. *Working Notes of the 5th International Planning Competition*, 2006.

Emil Keyder and Hector Geffner. Heuristics for planning with action costs revisited. In *Proc. of the 18th European Conference on Artificial Intelligence (ECAI)*, volume 178, pages 588–592, 2008.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

Sandra Kiefer and Brendan D. McKay. The iteration number of colour refinement. In *47th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 168, pages 73:1–73:19, 2020.

Diederick P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artif. Intell.*, 195:222–248, 2013.

Thorsten Klößner and Jörg Hoffmann. Pattern databases for stochastic shortest path problems. In *Proc. of the 14th International Symposium on Combinatorial Search (SOCS)*, pages 131–135, 2021.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

Richard E. Korf. Linear-space best-first search. *Artif. Intell.*, 62(1):41–78, 1993.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NeurIPS)*, pages 1097–1105, 2012.

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *CoRR*, abs/1604.00289, 2016.

Pascal Lauer, Alvaro Torralba, Daniel Fiser, Daniel Höller, Julia Wichlacz, and Joerg Hoffmann. Polynomial-time in pddl input size: Making the delete relaxation feasible for lifted planning. In *Proc. of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

Pan Li, Yanbang Wang, Hongwei Wang, and Jure Leskovec. Distance encoding: Design provably more powerful neural networks for graph representation learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:4465–4478, 2020.

Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in neural information processing systems*, 31, 2018.

Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *Proc. of the 20th European Conference on Artificial Intelligence (ECAI)*, volume 242, pages 540–545, 2012.

Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *Proc. of the 31st AAAI Conference on Artificial Intelligence*, pages 3590–3596, 2017.

Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *Proc. of the 14th National Conference on Artificial Intelligence (AAAI)*, pages 748–754, 1997.

Tengfei Ma, Patrick Ferber, Siyu Huo, Jie Chen, and Michael Katz. Online planner selection with graph neural networks and adaptive scheduling. In *Proc. of the 34th AAAI Conference on Artificial Intelligence*, pages 5077–5084, 2020.

Lawrence Mandow and José-Luis Pérez-de-la-Cruz. Multiobjective a* search with consistent heuristics. *J. ACM*, 57(5):27:1–27:25, 2010.

Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski. Modeling organic chemistry and planning organic synthesis. In *Proc. of the 1st Global Conference on Artificial Intelligence (GCAI)*, volume 36, pages 176–195, 2015.

Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014.

Kenneth L. McMillan. *Symbolic model checking.* Kluwer, 1993.

Argaman Mordoch, Roni Stern, and Brendan Juba. Learning safe numeric action models. In *Proc. of the 37th AAAI Conference on Artificial Intelligence*, 2023.

Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

Christopher Morris, Gaurav Rattan, and Petra Mutzel. Weisfeiler and leman go sparse: Towards scalable higher-order graph embeddings. *Neural Information Processing Systems (NeurIPS)*, 33, 2020.

Christopher Morris, Matthias Fey, and Nils M Kriege. The power of the weisfeiler-leman algorithm for machine learning with graphs. *International Joint Conference on Artificial Intelligence (IJCAI)*, 2021a.

Christopher Morris, Yaron Lipman, Haggai Maron, Bastian Rieck, Nils M Kriege, Martin Grohe, Matthias Fey, and Karsten Borgwardt. Weisfeiler and leman go machine learning: The story so far. *arXiv e-prints*, pages arXiv–2112, 2021b.

Bernhard Nebel. The small solution hypothesis for MAPF on directed graphs is true. *CoRR*, abs/2210.04590, 2022.

Ian Parberry. A real-time algorithm for the ($n^2$-1)-puzzle. *Information Processing Letters*, 56(1):23–28, 1995. ISSN 0020-0190.

Judea Pearl. *Heuristics - intelligent search strategies for computer problem solving.* Addison-Wesley series in Artificial Intelligence. Addison-Wesley, 1984.

Nir Pochter, Aviv Zohar, and Jeffrey S. Rosenschein. Exploiting problem symmetries in state-based planners. In *Proc. of the 25th AAAI Conference on Artificial Intelligence*, 2011.

Ira Pohl. Bi-directional and heuristic search in path problems. Technical report, Stanford University, 1969.

Ira Pohl. *Practical and theoretical considerations in heuristic search algorithms.* University of California (Santa Cruz). Information Sciences, 1975.

Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. From non-negative to general operator cost partitioning. In *Proc. of the 29th AAAI Conference on Artificial Intelligence*, pages 3335–3341, 2015.

Daniel Ratner and Manfred K. Warmuth. Finding a shortest solution for the N × N extension of the 15-puzzle is intractable. In *Proc. of the 5th National Conference on Artificial Intelligence*, pages 168–172, 1986.

Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In *Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.

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

Mehdi Samadi, Ariel Felner, and Jonathan Schaeffer. Learning from multiple heuristics. In *Proc. of the 23rd AAAI Conference on Artificial Intelligence*, pages 357–362, 2008.

Ryoma Sato. A survey on the expressive power of graph neural networks. *arXiv preprint arXiv:2003.04078*, 2020.

Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Random features strengthen graph neural networks. In *Proc. of the 2021 SIAM International Conference on Data Mining (SDM)*, pages 333–341, 2021.

Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *Proc. of the Extended Semantic Web Conference (ESWC)*, Lecture Notes in Computer Science, pages 593–607, 2018.

Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement. In *Proc. of the 23rd International Conference on Automated Planning and Scheduling (ICAPS)*, 2013.

Jendrik Seipp, Florian Pommerening, Gabriele Röger, and Malte Helmert. Correlation complexity of classical planning domains. In *Proc. of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3242–3250, 2016.

Jendrik Seipp, Thomas Keller, and Malte Helmert. Saturated cost partitioning for optimal classical planning. *J. Artif. Intell. Res.*, 67:129–167, 2020.

Vishal Sharma, Daman Arora, Florian Geißer, Mausam, and Parag Singla. Symnet 2.0: Effectively handling non-fluents and actions in generalized neural policies for RDDL relational mdps. In James Cussens and Kun Zhang, editors, *Proc. of the 38th Conference on Uncertainty in Artificial Intelligence, (UAI)*, volume 180, pages 1771–1781, 2022.

William Shen. Learning heuristics for planning with hypergraph networks. Bachelor's thesis, The Australian National University, 2019.

William Shen, Felipe W. Trevizan, Sam Toyer, Sylvie Thiébaux, and Lexing Xie. Guiding search with generalized policies for probabilistic planning. In *Proc. of the 12th International Symposium on Combinatorial Search (SOCS)*, pages 97–105, 2019.

William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proc. of 30th International Conference on Automated Planning and Scheduling (ICAPS)*, 2020.

Alexander Shleyfman, Michael Katz, Malte Helmert, Silvan Sievers, and Martin Wehrle. Heuristics and symmetries in classical planning. In *Proc. of the 29th AAAI Conference on Artificial Intelligence*, pages 3371–3377, 2015.

Silvan Sievers, Gabriele Röger, Martin Wehrle, and Michael Katz. Theoretical Foundations for Structural Symmetries of Lifted PDDL Tasks. In *Proc. of 29th International Conference on Automated Planning and Scheduling (ICAPS)*, volume 29, pages 446–454, 2019.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.

Tom Silver, Rohan Chitnis, Aidan Curtis, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Planning with learned object importance in large problem instances using graph neural networks. In *Proc. of the 35th AAAI Conference on Artificial Intelligence*, pages 11962–11971, 2021.

John K. Slaney and Sylvie Thiébaux. Blocks world revisited. *Artif. Intell.*, 125(1-2): 119–153, 2001.

Simon Staahlberg, Blai Bonet, and Hector Geffner. Learning generalized policies without supervision using gnns. In *Proc. of the 19th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2022a.

Simon Staahlberg, Blai Bonet, and Hector Geffner. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In *Proc. of 32nd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 629–637, 2022b.

Roni Stern and Brendan Juba. Efficient, safe, and probably approximately complete learning of action models. In *Proc. of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4405–4411, 2017.

Roni Stern, Tamar Kulberis, Ariel Felner, and Robert Holte. Using lookaheads with optimal best-first search. In *Proc. of the 24th AAAI Conference on Artificial Intelligence*, 2010.

Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3 (1):1–22, 1976.

Alvaro Torralba and Cosmina Croitoru. Planning systems and the IPC. University Lecture, Saarland University, 2019. Available online at: https://fai.cs.uni-saarland.de/teaching/winter18-19/planning-material/planning21-planning-systems-and-the-ipc-post-handout.pdf, last accessed on 05.05.2023.

Álvaro Torralba, Carlos Linares López, and Daniel Borrajo. Abstraction heuristics for symbolic bidirectional search. In *Proc. of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3272–3278, 2016.

Sam Toyer, Felipe W. Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *Proc. of the 32nd AAAI Conference on Artificial Intelligence*, pages 6294–6301, 2018.

Sam Toyer, Sylvie Thiébaux, Felipe W. Trevizan, and Lexing Xie. Asnets: Deep learning for generalised planning. *J. Artif. Intell. Res.*, 68:1–68, 2020.

Felipe Trevizan, Sylvie Thiébaux, P. Santana, and B. Williams. I-dual: Solving Constrained SSPs via Heuristic Search in the Dual Space. In *Proc. of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017a.

Felipe W. Trevizan, Sylvie Thiébaux, and Patrik Haslum. Occupation measure heuristics for probabilistic planning. In *Proc. of the 27th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 306–315, 2017b.

Felipe W. Trevizan, Sylvie Thiébaux, and Patrik Haslum. Operator counting heuristics for probabilistic planning. In *Proc. of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5384–5388, 2018.

Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

Alan M. Turing. Computing machinery and intelligence. *Mind*, LIX(236):433–460, 1950.

Karthik Valmeekam, Alberto Olmo Hernandez, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (A benchmark for llms on planning and reasoning about change). *CoRR*, abs/2206.10498, 2022.

Karthik Valmeekam, Sarath Sreedharan, Matthew Marquez, Alberto Olmo Hernandez, and Subbarao Kambhampati. On the planning abilities of large language models (A critical investigation with a proposed benchmark). *CoRR*, abs/2302.06706, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *International Conference on Learning Representations (ICLR)*, 2017.

Petar Velickovic, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *8th International Conference on Learning Representations (ICLR)*, 2020.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019.

Qing Wang, Dillon Ze Chen, Asiri Wijesinghe, Shouheng Li, and Muhammad Farhan. $\mathscr{N}$-WL: A new hierarchy of expressivity for graph neural networks. In *11th International Conference on Learning Representations (ICLR)*, 2023.

Asiri Wijesinghe and Qing Wang. A new perspective on" how graph neural networks go beyond weisfeiler-lehman?". In *International Conference on Learning Representations*, 2022.

Christopher Makoto Wilt and Wheeler Ruml. Building a heuristic for greedy search. In *Proc. of the 8th Annual Symposium on Combinatorial Search (SOCS)*, pages 131–140, 2015.

David H Wolpert and William G Macready. No free lunch theorems for search. Technical Report 05-010, Santa Fe Institut, 1995.

Fan Xie, Martin Müller, and Robert Holte. Adding local exploration to greedy best-first search in satisficing planning. In *Proc. of the 28th AAAI Conference on Artificial Intelligence*, pages 2388–2394, 2014.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? *8th International Conference on Learning Representations (ICLR)*, 2020.

Gal Yehuda, Moshe Gabel, and Assaf Schuster. It's not what machines can learn, it's what we cannot teach. In *Proc. of the 37th International Conference on Machine Learning, (ICML)*, volume 119, pages 10831–10841. PMLR, 2020.

Sung Wook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *J. Mach. Learn. Res.*, 9:683–718, 2008.

Jiaxuan You, Rex Ying, and Jure Leskovec. Position-aware graph neural networks. In *International Conference on Machine Learning (ICML)*, volume 97, pages 7134–7143. PMLR, 2019.

Jiaxuan You, Jonathan Gomes-Selman, Rex Ying, and Jure Leskovec. Identity-aware graph neural networks. In *Proc. of the 35th AAAI Conference on Artificial Intelligence*, 2021.

Muhan Zhang and Pan Li. Nested graph neural networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 34, 2021.

Lingxiao Zhao, Wei Jin, Leman Akoglu, and Neil Shah. From stars to subgraphs: Uplifting any gnn with local structure awareness. *International Conference on Learning Representations (ICLR)*, 2022.

Rong Zhou and Eric A. Hansen. Parallel structured duplicate detection. In *Proc. of the 22nd AAAI Conference on Artificial Intelligence*, pages 1217–1224, 2007.

Yichao Zhou and Jianyang Zeng. Massively parallel A* search on a GPU. In *Proc. of the 29th AAAI Conference on Artificial Intelligence*, pages 1248–1255. AAAI Press, 2015.