The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Engineering, Computing and Cybernetics (CECC)

Integrating Learning and Planning by Exploiting Action Structures

— Honours project (S1/S2 2024)

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

By:
Ryan Xiao Wang

Supervisor:
Felipe Trevizan

October 2024

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the Academic Integrity Rule;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course 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.

October, Ryan Xiao Wang

Acknowledgements

I am deeply grateful to my supervisor, Felipe Trevizan, for his continuous support, guidance, and encouragement throughout my honours year. Felipe has given me incredibly useful advice and pointers, and he has always supported me with the freedom to explore my research interests. His feedback on various drafts and ideas has consistently been invaluable, helping me refine my work and bring out the best in my research. I am also thankful for the ample computing resources provided through Felipe, including access to the Gadi supercomputer.

I would also like to thank my previous supervisor, Sylvie Thiébaux, for her support in my earlier research and her continued encouragement of my success. Sylvie generously funded my attendance at an international research conference during my honours year, which greatly broadened my knowledge and network in the field.

Lastly, I would like to thank my family for their unwavering support and encouragement, which has been my foundation throughout this journey.

Abstract

Artificial intelligence research is focused on developing intelligent systems that can automatically learn from data and make decisions. Tremendous progress has been made in the former, with deep learning systems achieving remarkable success and affecting many aspects of our lives. As a specific case of the latter, planning systems are capable of reliable long-horizon decision making in complex environments. Recently, strong interest has emerged in using learning methods to aid planning systems, with various approaches that achieve different levels of success. However, limited attention has been paid to how to adapt planning systems to make the best use of learning methods.

This thesis builds planning systems that are designed to work well with learning methods to allow for more effective decision making. We exploit the inherent relational structure of planning actions that is ignored by dominant planning systems. Through this, we introduce an alternative search space for planning that enables a more focused and efficient search. We discuss how this ultimately allows planning systems to receive more guidance from learning systems, and for learning systems to have more information to work with. We show how to extend any existing planning heuristic to work with our new search space, and how to learn new heuristics that are specifically designed for it. This way, we are able to build a learning-for-planning system, LazyLifted, where the planning component and learning component are designed for each other.

To evaluate LazyLifted, we use both existing competition benchmarks and new benchmarks designed to challenge planning systems with high branching factors. Our results show that LazyLifted outperforms existing state-of-the-art learning for planning systems. Furthermore, LazyLifted outperforms the state-of-the-art planning system LAMA-first, the first learning-for-planning system to do so.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Learning, planning, and learning for planning

The grand challenge of artificial intelligence (AI) is to develop systems that demonstrate human-like intelligence. AI research can be divided into two primary paradigms: learners and solvers, and categorised by Geffner (2018). Learners aim to infer knowledge from data and experiences without an explicit world model, while solvers focus on problem-solving based on a structured world model.

In the last decade, learners, particularly those based on deep learning, have achieved remarkable success and become a central focus in computer science. Advances in computational power and methodologies have powered breakthrough systems such as AlphaGo for the board game Go (Silver et al., 2016) and ImageNet for computer vision (Krizhevsky, Sutskever, and Hinton, 2012). More recently, large language models (LLM) have enabled interactive agents that offer experiences approaching human-like interaction. However, these systems still have limitations, including a lack of guarantees in their outputs, poor performance on out-of-distribution inputs, and limited transparency.

Solvers, particularly planners, are given a structured task description and seek to generate a plan — typically a sequence of actions — to achieve the specified goal (Geffner and Bonet, 2013). Planning systems are generally robust and reliable, promising to solve tasks provided unlimited time and memory. The core of planning research is to develop planning systems that support expressive description of planning tasks, such as uncertainty and time, and can solve them efficiently with high quality plans. The competing goals of expressiveness, efficiency, and quality means there is no one-size-fits-all solution to planning — a plethora of planning methodologies have been developed, each achieving a different trade-off (Helmert, 2006; Kurniawati, Hsu, and Lee, 2008; Scala et al., 2016; Trevizan et al., 2016).

The success of learners has made the idea of integrating them into planning highly

appealing. In recent years, a variety of approaches have emerged in the realm of learning for planning. Roughly categorising, these approaches either learn to directly plan (Toyer et al., 2020; Ståhlberg, Bonet, and Geffner, 2022a,b; Drexler, Seipp, and Geffner, 2023; Wang and Thiébaux, 2024), or learn to indirectly help or guide planners (Gnad et al., 2019; Shen, Trevizan, and Thiébaux, 2020; Chen, Thiébaux, and Trevizan, 2024; Hao et al., 2024). However, no system that uses learning in a significant way has yet to demonstrate itself as competitive with non-learning planners (Taitler et al., 2024). While this status is due in part to weaknesses in learning systems themselves, it also reflects the decades of accumulated development and optimisation in non-learning planners. For example, GOOSE (Chen, Trevizan, and Thiébaux, 2024), a state-of-the-art learning-for-planning system, learns better heuristic functions than those from non-learning methods. Yet it is still slightly inferior to the state-of-the-art non-learning planner, LAMA, due to other techniques employed by LAMA and the speed of evaluating the learned heuristic.

1.2 Planners for learners

In this thesis, we focus on classical planning, which features deterministic, fully observable environments and a discrete state space. The canonical approach for classical planning is search in the state space of the planning problem, guided by a heuristic function. GOOSE simply replaces the heuristic function from one obtained by non-learning methods to a learned heuristic. In this case, the learner provides a score to each successor state of a planning state while the planners uses these scores to decide how to proceed in the search.

When learners help planners like in GOOSE, in an ideal world, one would imagine a much more integrated, cooperative system. For example, the learner could help the planner gradually narrow down from a list of actions, rather than scoring at once all the actions' resulting states. This would see more involvement from the learner in the planning process, while at the same time providing the learner with more information to work with. The aim of this thesis is to design a planner and learner that work together more effectively, with the planner being better integrated with the learner and the learner utilising more information from the planner.

1.3 Contributions

This work focuses on building search-based planning and learning systems with deep integration. Our ultimate goal is to efficiently solve large classical planning problems. The resulting learning-for-planning system, LazyLifted, features:

1. *Fine-grained search.* We introduce a novel formulation of the classical planning problem into a search problem called *partial space*. We do so by exploiting the relational structure of planning actions. Searching in the partial space involves a reduced branching factor and more frequent opportunities for fine-grained heuristic guidance, when compared to traditional state space search. Under good heuristic

guidance, this ultimately results in a more focused and efficient search. Although designed for them, partial space search is not restricted to the use of learned heuristics or classical planning.

2. *Graph representations.* We design novel graph representations of search nodes of partial space search. These graphs represent different ways of interpreting what these nodes mean in the context of search. They are designed to be used to ultimately produce a feature vector representing the search nodes.

3. *Learned actions set heuristics.* Using our graph representations and the resulting feature vectors, we adapt and extend existing methods to learn action set heuristics. Unlike typical heuristics used in planning, which evaluate a single planning state, action set heuristics evaluate a set of actions on a state. This is akin to evaluating all the possible successor states led to by these actions. We learn these action set heuristics to guide partial space search. We are able to use a number of training plans on a domain to train an informed and fast heuristic function designed for partial space search.

4. *Extension of non-learning heuristics.* Although our focus is on learning, we also show that any traditional state space heuristic can be extended to an action set heuristic. This allows us to evaluate the merits of partial space search alone by using an efficient extension of the traditional h^{FF} heuristic.

To evaluate LazyLifted, we compare against a number of state-of-the-art learning and non-learning planners. We use the benchmarks from the International Planning Competition 2023 Learning Track (Taitler et al., 2024) and extend it with high branching factor problems. We show that LazyLifted is competitive with our baselines on normal classical planning tasks, and outperforms our baselines on hard-to-ground tasks. Ultimately, we show that LazyLifted is overall competitive with non-learning planners, the first learning-for-planning system to do so.

In additional to the main contributions, we have a number of additional contributions. Specifically,

1. *The LazyLifted planning system.* Typically, planning research is implemented on top of existing planning systems. However, we found implementing partial space search and action set heuristics requires significant changes to the underlying planner. As such, we instead developed a new planner from the ground up using the Rust programming language and following modern best practices. This planner, LazyLifted, is a lifted planner that is capable of working with large hard-to-ground planning tasks. We based its design on the existing state-of-the-art in lifted planning, namely Powerlifted (Corrêa et al., 2020). We implemented LazyLifted such that it is more performant in terms of search speed than Powerlifted. The LazyLifted codebase is also designed to be easy to maintain, extend, and use. Altogether, LazyLifted is a production ready planner that we hope can see real world use and be a base planner where future planning research can be implemented on

top of.

2. *High branching factor benchmarks.* Prior to our work, there was no organised collection of such benchmarks to our knowledge. We developed a set of benchmark planning tasks for high branching factor planning problems. We did so by following the rough design of the benchmarks from the International Planning Competition 2023 Learning Track. Our benchmark set, although not large, includes a variety of problems. We hope this benchmark set can help evaluate future research in this area more comprehensively.

3. *Rank2Plan.* As part of our efforts to learn action set heuristics, we extended the methods proposed in Dedieu, Mazumder, and Wang (2022) for efficiently training L1-regularised Support Vector Machines (SVM) to RankSVMs for planning. We implemented our extension in an open-source Python package called Rank2Plan. We found that this significantly reduced the training time and memory of our heuristics. This package is generally useful for learning planning heuristics, and we hope it can be used by the planning community.

1.4 Structure of this thesis

To clearly and thoroughly present our contributions, this thesis is structured as follows:

- Chapter 2 provides background information on classical planning and learning for planning. We formalise the planning task and explain the main ideas and concepts in heuristic search, the current state-of-the-art in solving classical planning problems. We also introduce the concept of learning for planning and the current state-of-the-art in this area, which is based on graph representations and kernels with classical machine learning techniques.

- Chapter 3 introduces our new search space formulation, partial space search, in detail. We introduce the concept of a partial action, which represent sets of actions, and the natural tree structure of partial actions. We use this tree structure to define partial space search. Moreover, we also discuss how partial space search is more efficient than the traditional state space search, and what the implications are for learning methods.

- Chapter 4 introduces the notion of action set heuristics, which evaluate sets of actions on a state. We discuss how action set heuristics are capable of guiding partial space search. We show how to automatically translate any traditional state space heuristic into an action set heuristic, and how to do this efficiently, using the FF heuristic as an example. More importantly, we discuss graph representations that can be used to learn action set heuristics, and how we adapt and improve existing learning methods to learn these heuristics.

- Chapter 5 evaluates our contributions. Here we introduce the engineering side of our new LazyLifted planner and the Rank2Plan library for efficient training.

We also introduce our new set of benchmarks for high branching factor planning problems. We then evaluate LazyLifted on competition benchmarks from the International Planning Competition 2023 learning track and our new benchmarks We provide a detailed and thorough analysis of the results. We ultimately show that our contributions achieve their design goals. Specifically, we show that partial space search is effective on high branching factor tasks, our automatically translated action set heuristics perform as intended, and our learned action set heuristics are the new state-of-the-art learned heuristics. We also show that our contributions work together to outperform the state-of-the-art planner LAMA-first.

- Chapter 6 surveys existing research works relevant to our contributions. We discuss where our work fits in the existing literature and how it builds upon and differs from previous works.
- Chapter 7 concludes the thesis. We summarise our contributions and discuss the implications and limitations of our work. We also discuss future research directions that can build upon our work.

Chapter 2

Background

In this chapter we will introduce the necessary background to explain our work. We will also discuss adjacent topics that are relevant to our work and the state-of-the-art in the field. We will start by formalising the planning task in Section 2.1, followed by a discussion on heuristic search approaches in Section 2.2. Then, we will discuss the state-of-the-art in learning for planning in Section 2.3 by discussing how graphs are used for learning planning heuristics.

2.1 The Planning Task

In this section we introduce the classical planning task, including a commonly used formalism based on the Planning Domain Definition Language (PDDL).

In classical planning, one seeks to find a sequence of actions that transition from a given initial state to a goal state, where the environment is fully observable and deterministic, and the state and action spaces are discrete (Geffner and Bonet, 2013).

Definition 1 (Planning task)**.** *A* planning task *is a tuple* $\Pi = \langle \mathbb{S}, s_0, G, \mathbb{A}, \tau, c \rangle$ *where:*

- $\mathbb{S}$ *is a set of states,*
- $s_0 \in \mathbb{S}$ *is the initial state,*
- $G \subseteq \mathbb{S}$ *is a non-empty set of goal states,*
- $\mathbb{A}$ *is a set of actions where for each state* $s \in \mathbb{S}$*,* $\mathbb{A}_s$ *is the set of applicable actions in* s*.*
- $\tau : \mathbb{S} \times \mathbb{A} \to \mathbb{S} \cup \{\bot\}$ *is the transition function, where if* $a \in \mathbb{A}_s$ *then* $\tau(s, a) \in \mathbb{S}$ *is the resulting state of applying action* a *in state* s*, and* $\bot$ *if* $a \notin \mathbb{A}_s$*.*
- $c : \mathbb{S} \times \mathbb{A} \to \mathbb{R}_{\geq 0}$ *is the cost function, where* $c(a)$ *is the cost of applying action* a*.*

A *plan* for this task is a sequence of actions $\pi = a_1, \ldots, a_n$ such that their iterative execution starting at s_0 leads to a goal state in G. Specifically, this means there exists a sequence of states $s_0, \ldots, s_n$ such that for each i from 1 to n, a_i is applicable in the state s_{i-1}, and $\tau(s_{i-1}, a_i) = s_i$, with $s_n \in G$. The cost of a plan is given by $c(\pi) = \sum_{i=1}^{n} c(s_{i-1}, a_i)$, and the *trace* of a plan π is the sequence of states visited, namely $s_0, \ldots, s_n$. We say a planning task is *solvable* if there exists a plan for it, and *unsolvable* otherwise. An *optimal plan* is a plan with the minimum cost among all plans for a planning task.

2.1.1 The Planning Domain Definition Language

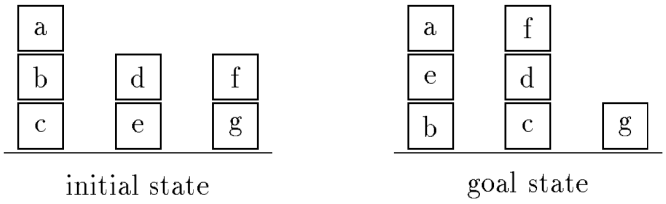
Directly working with the planning task as defined in Definition 1 often yields infeasibly large state and action spaces. Rather than directly working with it, the planning community has developed formalisms for more compactly representing planning tasks. The Problem Domain Definition Language (PDDL) is a commonly used modelling language for describing planning tasks (Haslum et al., 2019). Here, instead of introducing the syntax of PDDL, we focus instead on the underlying formalism for the planning task, which is often called the first-order or lifted planning task (Geffner and Bonet, 2013). In particular, we focus on the STRIPS fragment of PDDL for this work.

Definition 2 (Lifted Planning Task)**.** *A* lifted planning task *is a tuple* $\Pi = \langle D, I \rangle$ *where* D *is the planning* domain *and* I *is the planning* instance*. The domain* D *is a tuple* $\langle \mathcal{P}, \mathcal{A} \rangle$ *where* $\mathcal{P}$ *is a set of* predicates *and* $\mathcal{A}$ *is a set of* action schemas*. The instance* I *is a tuple* $\langle O, s_0, G \rangle$ *where* O *is a set of* objects*,* s_0 *is the initial state, and* G *is the goal condition.*

Each predicate $P \in \mathcal{P}$ is a symbol that can be *instantiated* or *grounded* with objects from O to form propositions of the form $P(o_1, \ldots, o_n)$, where $o_1, \ldots, o_n$ are objects from O and $n \in \mathbb{Z}_{\geq 0}$ is the *arity* of P. Through grounding, $\mathcal{P}$ and O define the set of all propositions (also called *atoms*), which encode a state space $\mathbb{S}$ where each state is an assignment of boolean values to each proposition. Often, states are viewed as sets of propositions that are true in the state.

The initial state s_0 is simply an element of $\mathbb{S}$, while the goal condition G is a set of propositions that must be true in the goal state. Any state that contains G is a goal state.

The action schemas $\mathcal{A}$ define the state transition system. Each action schema $A \in \mathcal{A}$ is a tuple $\langle \Delta(A), \text{pre}(A), \text{add}(A), \text{del}(A) \rangle$ where $\Delta(A)$ is the parameters of A in a fixed order. The *precondition* $\text{pre}(A)$, *add effect* $\text{add}(A)$, and *delete effect* $\text{del}(A)$ are sets of propositions instantiated from the parameters and objects $\Delta(A) \cup O$. Action schemas are instantiated by substituting the parameters with concrete objects in O, yielding a ground action a whose precondition, add effect, and delete effects are sets of atoms. The set of actions $\mathbb{A}$ is the set of all possible action instantiations from $\mathcal{A}$. The preconditions $\text{pre}(a)$ define the set of propositions that must be true in the state for the action to be applicable, while $\text{add}(a)$ and $\text{del}(a)$ define the set of propositions that are added



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 2.1: An example Blocksworld planning task with an example plan, taken from Slaney and Thiébaux (2001).

and deleted from the state, respectively, when the action is executed. Specifically, the application of an action a in a state s produces a successor state $s' = (s \setminus \text{del}(a)) \cup \text{add}(a)$.

Unlike the basic STRIPS fragment of PDDL, which assumes all actions have cost 1, for this work we also use $c : \mathbb{A} \to \mathbb{R}_{\geq 0}$ to denote the cost of a particular ground action in the lifted planning task. In actual PDDL, this is modelled through an extension which extends action schemas to also model the cost of the action.

The lifted planning task allows for a much more compact representation. Consider the common example planning task, Blocksworld, where given a set of blocks, the goal is to move them from a starting stacking configuration to a goal stacking configuration, as shown in Figure 2.1. To describe Blocksworld using the formulation from Definition 1, one would need to enumerate all possible block configurations, which is infeasible when there are many blocks. In comparison, using the lifted formalism, one can define the objects as the blocks, predicates to encode the relationship between the blocks, and action schemas for moving blocks. This allows for a much more compact representation of the task.

The lifted planning task also provides much more structure to the planning task. For example, the action schemas provide a structured view of the action space, where each action schema defines a set of similar actions applied to different objects. This structure can be exploited by planning algorithms for more effective reasoning, as is done in our work. Additionally, its separation of the domain and instance allows for the domain to be defined once and reused for multiple instances. This is useful for planning problems with similar structures but with slightly different initial states and goals with different objects. This domain and instance structure is also useful for learning-based approaches, where domain knowledge can be learned and reused across different instances (Toyer et al., 2020; Shen, Trevizan, and Thiébaux, 2020; Ståhlberg, Bonet, and Geffner, 2022b).

2.1.2 State-of-the-art in planning

When solving a planning task, one typically aims to either quickly find a plan that reaches the goal regardless of cost, or to find an optimal plan. The former is called *satisficing planning*, while the latter is called *optimal planning*. Planners for these two types of planning tasks are called *satisficing planners* and *optimal planners*, respectively.

In the world of satisficing planning, the current state-of-the-art planning system is LAMA (Richter and Westphal, 2010). LAMA is a satisficing planner that uses heuristic search (see Section 2.2) with multiple heuristics and other techniques to achieve high performance on a wide range of planning tasks. Strictly speaking, LAMA is an any-time planner that produces a plan, and continually seeks to find plans of better quality until terminated, while LAMA-first is a variant that terminates after finding the first plan. Its status as the state-of-the-art is clear from its performance in the International Planning Competition (IPC) 2023 satisficing track (Taitler et al., 2024), where the goal is to not just find plans, but also high quality plans. Here, it is used as a baseline, outperforming most participants and achieving a very similar performance to the best participants. Additionally, LAMA-first is also used as a baseline planner in the agile track, where the goal is to find plans quickly, and it outperforms all participants in this track as well.

For optimal planning, the state-of-the-art planner is Scorpion (Seipp, 2023). Similar to LAMA, it uses heuristic search with a variety of techniques to achieve high performance while guaranteeing optimality. Scorpion comes second in the IPC 2023 optimal track, losing only to a planner that intelligently combines multiple other planners together (Drexler et al., 2023).

2.2 Heuristic Search

Planning tasks resemble a search problem on a graph where the nodes are the planning states and the edges are the actions that transition between states. In this view, planning mirrors the graph path finding problem where the edge weights are action costs. Unsurprisingly, search in this graph is the de facto method for solving planning tasks. However, the state space graph is typically so large that it cannot even be explicitly encoded in memory. To make the search tractable, heuristic functions are used to guide the search algorithm to traverse through the graph more effectively. In this section, we will first introduce basic definitions of what a heuristic function is. We will then discuss common search algorithms that use these heuristics to solve planning tasks in Section 2.2.1. We will next discuss methods for exploring the state space graph without explicitly encoding it in memory in Section 2.2.2. Lastly, we discuss the well-known heuristics used in planning in Section 2.2.3.

It is important to note that most things we introduce in this section are specific to dominant search space for classical planning — state space search. Alternative search spaces also exist for planning, such as plan space search, where one searches through a space of partially specified plans (Penberthy and Weld, 1992). A main contribution of

this thesis is a new search space, which we will introduce later in Chapter 3. In the rest of this chapter, we will avoid repetitively emphasising state space search and just refer to it as search. However, later on we will make it clear when we are referring to state space search.

A *heuristic* function (or simply, a heuristic) for a planning problem is a function of the form $h : \mathbb{S} \to \mathbb{R} \cup \{\infty\}$, where $\mathbb{S}$ is the state space of the planning task. The value ∞ can be used to denote *deadends*, states where no plan can reach the goal. A heuristic function is estimate of how close a state is to the goal, where a lower value indicates the state is closer to the goal. Typically, heuristic functions aim to estimate the cost to transition from a state to a goal state (cost-to-go).

The optimal heuristic h^* assigns to each state s to cost of the optimal plan from s to a goal state. An *admissible* heuristic is one that never overestimates the cost to reach a goal state. More precisely, a heuristic h is admissible if for all $s \in \mathbb{S}$, $h(s) \leq h^*(s)$. A heuristic is *goal-aware* if it assigns the value 0 to all goal states. A heuristic is *consistent* if it is goal-aware and $h(s) \leq c(s, a)+h(\tau(s, a))$, where a is an applicable action in state s. This means that consistent heuristic respect action costs, and never overestimate them. A consistent heuristic is always admissible, although the converse is not true. Lastly, a heuristic h dominates another heuristic h' is $h(s) \geq h'(s)$ for all $s \in \mathbb{S}$. Intuitively, for admissible heuristics which always underestimate the cost-to-go, a higher heuristic value indicates less underestimation, so one would prefer the dominating heuristic.

2.2.1 Heuristic search algorithms

A heuristic search algorithm uses guidance from a heuristic function to explore a graph. Most such algorithms maintain a frontier of nodes to explore, and expand nodes from this frontier based on the heuristic value of the nodes. A skeleton of a generic search algorithm is shown in Algorithm 1. The most commonly used heuristic search algorithms in planning are A* and Greedy Best First Search (GBFS), as well as their variations. In this section we will simply refer to heuristic search algorithms as search algorithms.

The A* algorithm provides theoretical guarantees that makes its use prevalent in optimal planning (Pearl, 1984). We use g to denote the function that maps each node to the cost of the shortest known path from the start node to that node so far in the search. Given a heuristic h, the A* algorithm assigns to each node n a f-value that is simply the sum of $h(n)$ and $g(n)$, and uses a frontier that is a priority queue ordered by f-values. That is, A* always expands the node with the lowest f-value. Note that in lines 11-12 of Algorithm 1, A* reopens nodes if it finds a shorter path to a node. If h is admissible, then plans found by A* are optimal. If h is additionally consistent, then A* is guaranteed to never reopen nodes. Furthermore, given a consistent heuristic, A* is also guaranteed to expand the least number of nodes among all algorithms of a similar form using the same heuristic.

There are also a number of variations of A* to achieve different performance characteristics. To reduce space complexity, one can employ iterative deepening A*, which limits

Algorithm 1: Skeleton of a generic search algorithm

```
Data: Planning problem ⟨𝕊, s_o, G, 𝔸, τ, c⟩; heuristic h.
1  OPEN ← ∅
2  s.closed ← ⊥, ∀s ∈ 𝕊
3  OPEN.push(s_0, f(s_0))            /* f depends on the search algorithm */
4  while OPEN ≠ ∅ do
5      s ← OPEN.popFront()
6      s.closed ← ⊤
7      if s ∈ G then
8          return plan from s
9      for a ∈ 𝔸_s do
10         t ← τ(a, s)
11         if t.closed = ⊤ then
12             continue              /* Some algorithms may also reopen t */
13         OPEN.push(t, f(t))
14 return no solution
```

the depth of the search tree (through f-value) and gradually restarts and relaxes the limit until a solution is found (Korf, Reid, and Edelkamp, 2001). To improve search time by sacrificing optimality, weighted A* uses a weight w to redefine the f-value as $f(n) = g(n) + w \cdot h(n)$, where $w > 1$. This results in a more greedy search that prioritises nodes with low heuristic values more. Given an admissible heuristic, weighted A* ensures that plans it finds has a cost that is bound by w times the cost of the optimal plan.

Greedy Best First Search (GBFS) can be considered as weighted A* taken to the extreme. GBFS uses a f-value that is simply the heuristic value $h(n)$, such that it always expands the node with the least heuristic value. GBFS provides no optimality guarantees, and is commonly used in satisficing planning for fast search time.

The algorithms described so far all perform *eager evaluation*, that is, the heuristic value of a node is computed as soon as it is added to the frontier. However, such evaluation can also be deferred to when the node is expanded, resulting in *lazy evaluation* (Richter and Helmert, 2009). Since a value of a node is still required to determine the order of expansions, the value of its parent is used in the frontier. This can provide a speedup in search time, especially when the number of nodes generated during search is much larger than the number of nodes expanded. However, lazy evaluation means less useful information is used to determine the order of expansions, resulting in a trade-off between number of heuristic evaluations and number of nodes expanded.

There are also additional techniques that aim to provide more information to the search algorithm beyond heuristics. To help guide the search algorithm to interesting, unex-

plored parts of the state space, Lipovetzky and Geffner (2012) proposed Width-Based Search, which employs a novelty measure. Similarly, *Preferred operators* seek to prioritise some actions (operators) over others. Search algorithms can take advantage of preferred operators by only expanding nodes from preferred operators. Alternatively, one can perform a dual-queue approach, one for preferred operator nodes, and the other for all the nodes, and dynamically adjusting which queue to use based on which one has been more helpful (Richter and Helmert, 2009; Corrêa and Seipp, 2022). In particular novelty measures can be used to construct preferred operators (Corrêa and Seipp, 2022).

2.2.2 Grounded versus lifted search

As we discussed, the state space graph of a planning task is typically too large to be explicitly encoded in memory. This necessitates techniques to explore the state space without needing to encode the entire graph.

Here we focus our description on the translation process of the Fast Downward planning system (Helmert, 2006). Fast Downward has been the state-of-the-art classical planning system for the last two decades, with various techniques being implemented in it over time. Given a lifted planning task, Fast Downward first grounds the task, i.e., computing the set of all atoms and actions relevant to in the problem. It then analyses the set of atoms to translate groups of these binary variables to single multivalued variables. Afterwards, it performs various forms of additional analysis to compute structures useful for search, such as a successor generator data structure for computing the set of applicable actions in a state. This way, grounding allows for efficient exploration of the state space without needing to explicitly encode the state space graph in memory.

For large planning tasks, grounding can still be memory and time intensive. Lifted planning, where grounding is not performed, is particularly important for solving these hard-to-ground (HTG) planning tasks. Corrêa et al. (2020) applied techniques from database theory to compute applicable actions in a planning state. Specifically, they represent each state as a database, and formulate the task of computing the applicable actions for a particular action schema as a query on this database. They then compute all the applicable actions by performing this query for all action schemas. This allows for exploring the state space of planning tasks without even needing to ground the task. However, the lifted approach is typically somewhat less efficient than grounded approaches, since grounding precomputes much of the work that the lifted approach must do on the fly.

2.2.3 Heuristics

Commonly, the most important factor in whether a heuristic search approach is successful is the heuristic. In particular, *domain-independent* heuristics, which can be applied to any planning domain, receive much of the research attention. This is in contrast to *domain-specific* heuristics, which are usually handcrafted for a particular domain and

hence limited in general utility. We focus this section on providing a survey of domain-independent heuristics.

The most naive heuristic is simply the zero heuristic, defined by $h(s) = 0$ for all states s, which provides no information at all. On the other side of the spectrum we have the perfect heuristic h^*, which maps every state to exactly its minimum cost to reach the goal, hence providing perfect information. Computing the perfect heuristic for a certain state s by definition is the same as solving the planning task with initial state s. More heuristics therefore lie between the zero and perfect heuristic in terms of informedness.

Examples of a simple domain-independent heuristic is the inadmissible goal counting heuristic h^{gc}, which maps states to the number of unachieved goals. The goal counting heuristic can be surprisingly effective due to being easy to compute and relatively well-informed. Still, we often require stronger heuristics to effectively solve many planning tasks.

Delete relaxation heuristics

A common approach for constructing heuristics, both domain-independent and domain-dependent, is *relaxation*. Relaxation heuristics are based on relaxing certain aspects of the original planning task such that they become easy to solve directly. Given a state s, they then map s to the cost of solving the relaxed problem starting at s optimally. This is particularly useful, as by making the problem easier to solve, they give underestimated costs and hence are admissible. Relaxation heuristics can also give estimated costs of solving the relaxed problem, which is not always admissible. This is particularly useful when even the relaxed problem is hard to solve optimally.

An early domain-independent relaxation, which is very much still useful today, is *delete relaxation*. Delete relaxation simply sets the delete effects of all action schemas to be empty. This way, atoms, once achieved, stay achieved forever, and similarly applicable actions stay applicable forever.

The *perfect delete relaxation heuristic*, h^+, simply computes the cost to optimally solve the delete relaxed problem from the input state. Its computation is NP-complete. Haslum, Slaney, and Thiébaux (2012) computed h^+ using *disjunctive action landmarks*. A disjunctive action landmark is a set of actions where the application of at least one of them is necessary for achieving the goal. The heuristic value of h^+ can be computed by iteratively solving minimum cost hitting set problems for increasing sets of disjunctive action landmarks.

The heuristics h^{add} and h^{max} are tractable alternatives to h^+ (Bonet and Geffner, 2001). On top of the delete relaxation, these two heuristics make the additional assumption on the independence of subgoals. They decompose the goal atoms of the delete relaxed problem into subgoals, recursively compute the cost to achieve subgoals independently, then aggregate them together. The two heuristics only differ in that h^{add} sums the subgoal costs together, while h^{max} takes the maximum subgoal cost. The additive ag-

gregation in h^{add} is pessimistic, assuming that all the subgoals are entirely independent, and no action applied in reaching one goal is of any use for reaching others. On the other hand, the maximum aggregation in $h^{\max}$ is optimistic, assuming that all the subgoals are reached just by reaching the hardest subgoal. It has been shown $h^{\max}$ is admissible and dominated by h^+, while h^{add} is inadmissible and dominates h^+. Their exact definitions are given in Definition 3. Here we use the version by Keyder and Geffner (2008) that considers action costs.

Definition 3 (h^{add} and $h^{\max}$)**.** *Let $\Pi = \langle\langle \mathcal{P}, \mathcal{A}\rangle, \langle O, s_0, G\rangle\rangle$ be a lifted planning task, with $\mathbb{A}$ the set of all ground actions. The heuristics h^{add} and $h^{\max}$ are defined by $h^{\text{add}}(s) = h^{\text{add}}(s, G)$ and $h^{\max}(s) = h^{\max}(s, G)$, where*

$$h^{\text{add}}(s,g) = \begin{cases} 0, & \text{if } g \subseteq s \\ \min_{a\in\mathbb{A}, p\in\text{add}(a)} \left[c(a) + h^{\text{add}}(s, \text{pre}(a))\right], & \text{if } g = \{p\} \\ \sum_{p\in g} h^{\text{add}}(s,\{p\}), & \text{if } |g| > 1 \end{cases},$$

and

$$h^{\max}(s,g) = \begin{cases} 0, & \text{if } g \subseteq s \\ \min_{a\in\mathbb{A}, p\in\text{add}(a)} \left[c(a) + h^{\max}(s, \text{pre}(a))\right], & \text{if } g = \{p\} \\ \max_{p\in g} h^{\max}(s,\{p\}), & \text{if } |g| > 1 \end{cases}.$$

Recall that in Section 2.1.1 we said that states are often viewed as the set of propositions that are true in the state. Here, $g \subseteq s$ means that all propositions in g are true in s.

The heuristic h^+ can also be approximated using the h^{FF} heuristic, which is a better approximation than h^{add} (Hoffmann and Nebel, 2001).

Definition 4 (h^{FF})**.** *Let $\Pi = \langle\langle \mathcal{P}, \mathcal{A}\rangle, \langle O, s_0, G\rangle\rangle$ be a lifted planning task, with $\mathbb{A}$ the set of all ground actions, s the state to compute the heuristic for, and $\gamma : \mathbb{A} \to \mathbb{R}_{\geq 0}$ an achiever cost function. We assume the problem has been modified such that there is only one goal atom* `goal`*, with a special* `achieve-goal` *action, whose precondition is all the original goal atoms and effect is to add* `goal`*.*

For each atom p let a_p be an action with the least γ value in the set of all actions that achieve p, i.e., satisfy $p \in \text{add}(a)$. Let

$$\pi(p) = \begin{cases} \emptyset, & \text{if } p \in s \\ \{a_p\} \cup \bigcup_{q\in\text{pre}(a_p)} \pi(q), & \text{otherwise} \end{cases}.$$

If we fix a_p for all reachable atoms from s and the goal atoms are all reachable, we can recursively compute a unique solution to π. Then, $\pi(\texttt{goal})$ is a set of actions that can be sequenced to a plan to the delete relaxed problem, and the heuristic value $h^{\text{FF}}(s)$ is its cost.

The achiever cost function measures which action should be preferred for achieving a particular atom. Hoffmann and Nebel (2001) defined it based on $h^{\max}$ values, while

Keyder and Geffner (2008) did so using h^{add} values, namely $\gamma(a) = c(a)+h^{\text{add}}(s, \text{pre}(a))$. We use will use the latter definition.

It is worth noting that the delete relaxation heuristics introduced here, namely h^{add}, h^{max}, and h^{FF}, cannot be easily computed in the lifted planning case. We discuss their lifted computation later this section.

Abstractions, cost partitioning, and landmarks

The delete relaxation heuristics discussed so far, specifically h^{add} and h^{FF}, remain to this day some of the most effective heuristics for satisficing planning. Here we discuss briefly some domain-independent admissible heuristics for optimal planning. As we will discuss later this section, highly accurate admissible heuristics are not necessarily the best heuristics to use for satisficing planning.

Abstraction heuristics are computed from abstractions of planning tasks, which are, at a high level, simplifications of planning tasks. For the same reason that relaxation heuristics like h^+ that compute minimum costs are admissible, abstraction heuristics are also admissible. Pattern databases (PDBs) is an early abstraction heuristic that abstract away all but a small part of the planning task, leaving the resulting problem (pattern) easy to solve by just blind search (Edelkamp, 2001). Patterns are typically combined for the computation of the canonical heuristic, which requires that the individual patterns are orthogonal for admissibility (Haslum et al., 2007). This means that each action cannot affect multiple patterns. Alternatively, merge and shrink (M&S) abstractions allow constructing a single good abstraction by searching over the space of abstractions and merging them or shrinking them (Helmert, Haslum, and Hoffmann, 2007). M&S abstractions are shown to be the most general abstractions, where any abstraction can be represented as a M&S abstraction. Lastly, Cartesian abstractions, which iteratively refine abstractions by discovering counterexamples, offer an efficient and fine-grained refinement and generalise pattern databases (Seipp and Helmert, 2013).

Cost partitioning is a method for additively combining admissible heuristics such that the combined heuristic remains admissible. Specifically, given n admissible heuristics, cost partitioning produces n copies of the original planning problem. Each copy contains a modified cost function, which can differ between copies. The component heuristics are computed on their respective copy, and the resulting heuristic values are summed for the overall heuristic value. Katz and Domshlak (2008) showed that if for each action, the sum of its costs over all copies is a lower bound of the original cost, then the combined heuristic through cost partitioning is admissible. The distribution of the original action costs over the copies is called the cost partitioning, which obviously plays an important role in the quality of the resulting admissible heuristic. The h^{max} heuristic, which is often not informative due to being extremely optimistic, can be improved admissibly by applying cost partitioning to many copies of h^{max}, leading to the additive h^{max} heuristics.

We had previously discussed disjunctive action landmarks, which are sets of actions where one from each set must be applied to reach the goal. Let L be a set of actions and

L^+ their delete relaxation (i.e., with delete effects set to the empty set), we say L^+ is a s-landmark if it is a disjunctive action landmark from the state s in the delete relaxed problem. The elementary landmark heuristic for a set of actions L is a simple heuristic that maps each state s to the cost of the cheapest action in L if L^+ is a s-landmark, and otherwise to 0. They are clearly admissible, and the elementary landmark heuristics for disjunctive action landmarks in the delete relaxed problem can be combined admissibly through cost partitioning. Karpas and Domshlak (2009) showed that the optimal cost partitioning for this can be computed in polynomial time.

Helmert and Domshlak (2009) showed close relations between abstraction heuristics, additive $h^{\max}$ heuristics, and landmark heuristics. Specifically, they showed that these heuristics can often be compiled into each other in polynomial time. Their results yield the $h^{\text{LM-Cut}}$ heuristic, which is the result of first compiling $h^{\max}$ to a set of disjunctive action landmarks, then combining their elementary landmark heuristics using cost partitioning. Crucially, they showed that $h^{\text{LM-Cut}}$ is admissible and dominates $h^{\max}$.

Datalog and lifted heuristics

We had previously in Section 2.2.2 discussed the importance of lifted search for solving large, hard-to-ground planning tasks. A key challenge in lifted planning is the need by almost all heuristics discussed so far to ground the planning task. Recent developments have seen the use of Datalog, a logic programming language, to compute the delete relaxation heuristics efficiently in a lifted way (Corrêa et al., 2021). These lifted implementations are the state-of-the-art in lifted planning, and have competitive performance with their grounded counterparts when grounding is possible (Corrêa et al., 2022).

A *Datalog rule* r has the form $P \leftarrow Q_1, \ldots, Q_m$, for $m \geq 1$. Here, P is the head of the rule, denoted $\mathit{head}(r)$, and $Q_1, \ldots, Q_m$ are the body of the rule, denoted $\mathit{body}(r)$. Each term $P, Q_1, \ldots, Q_m$ is a predicate[1] with some number of argument variables. Given a set of constants $\mathcal{C}$, $\mathit{Ground}(r)$ is the set of all rules obtained by substituting variables in r with constants from $\mathcal{C}$.

A *Datalog program* $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ is made up of a set of facts (ground atoms) $\mathcal{F}$ and a set of rules $\mathcal{R}$. Given a Datalog rule $r \in \mathcal{R}$ where $r = P \leftarrow Q_1, \ldots, Q_m$ with variables $\mathit{vars}(r) = \{v_1, \ldots, v_n\}$, its semantics can be written as $r_\forall = \forall v_1, \ldots, v_n.\ Q_1 \wedge \ldots \wedge Q_m \rightarrow P$. We use $\mathit{Ground}(\mathcal{R})$ to denote $\bigcup_{r \in \mathcal{R}} \mathit{Ground}(r)$. The *canonical model* of the program $\mathcal{D}$ is the maximal set $\mathcal{M}$ of ground atoms such that $\mathcal{F} \cup \{r_\forall | r \in \mathcal{R}\} \models \mathcal{M}$. That is, the canonical model is the set of all ground atoms that can be derived from the facts and rules of the program. The canonical model of a Datalog program is unique (Abiteboul, Hull, and Vianu, 1995), and its computation is EXPTIME-complete (Dantsin et al., 2001).

A derivation of a ground atom p from a Datalog program $\mathcal{D}$ is a sequence of facts from $\mathcal{F}$ and ground rules from $\mathit{Ground}(\mathcal{R})$, where all the body atoms of each rule in the

[1] Note that Datalog predicates and atoms are not the same as planning predicates and atoms, even if their syntax and semantics are highly similar.

sequence appears earlier as facts or head of rules in the sequence, and where the last fact or head of rule is p. A derivation for a ground atom p is a proof that $p \in \mathcal{M}$.

The key to using Datalog for computing delete relaxation heuristics is model a lifted planning task as a Datalog program. Given a lifted planning task $\Pi = \langle\langle \mathcal{P}, \mathcal{A}\rangle, \langle O, s_0, G\rangle\rangle$ and a state s, Corrêa et al. (2021) encode the delete relaxation of Π, Π^+, as a Datalog program $\mathcal{D}_s = \langle \mathcal{F}, \mathcal{R}\rangle$. The facts $\mathcal{F}$ contains all ground atoms in s, while $\mathcal{R}$ contains rules that encode the action schemas $\mathcal{A}$. Specifically, for each action schema $A \in \mathcal{A}$ with parameters $\Delta(A)$ and precondition $\text{pre}(A) = \{Q_1, \ldots, Q_m\}$, $\mathcal{R}$ contains the action applicability rule

$$A\text{-applicable}(\Delta(A)) \leftarrow Q_1, \ldots, Q_m,$$

and for each add effect $P \in \text{add}(A)$, $\mathcal{R}$ contains the action effect rule

$$P \leftarrow A\text{-applicable}(\Delta(A)).$$

Helmert (2009) showed that the canonical model of $\mathcal{D}_s$ contains exactly the reachable ground actions and atoms from s in the delete relaxed problem Π^+. There, Helmert used it to ground the lifted problem by computing the canonical model incrementally, similar to a generalised Breadth First Search. Specifically, the algorithm starts with a queue $\mathcal{Q} = \mathcal{F}$ and $\mathcal{M} = \emptyset$. In each iteration, it pops a fact f from $\mathcal{Q}$. If f is already in $\mathcal{M}$, it moves to the next iteration. Otherwise, it adds f to $\mathcal{M}$ and computes the set of all ground rules r where $\mathit{body}(r) \subseteq \mathcal{M}$ and $f \in \mathit{body}(r)$, and adds the head of each such rule to $\mathcal{Q}$. This algorithm implicitly constructs a derivation for each atom in $\mathcal{M}$. Furthermore, for each atom p in $\mathcal{M}$, its *best achiever* is the rule that first added p to $\mathcal{Q}$. We refer to Helmert (2009) and Corrêa et al. (2022) for an explanation of preprocessing and optimisations techniques for making this algorithm efficient.

In order to compute delete relaxation heuristics from $\mathcal{D}_s$, Corrêa et al. (2021) modified it by adding a rule whose head is an auxiliary goal atom `goal`, and body are the original goal atoms. They also included a weight for each Datalog rule. In particular, they assume that all actions from the same action schema have the same cost, and assign the weight $c(A)$ to the action applicability rule generated for each action schema $A \in \mathcal{A}$, and the weight 0 to all other rules. They then modified the above incremental algorithm to capture rule weights. Specifically, they order facts in $\mathcal{Q}$ by a value v. They set $v(p) = 0$ for all initial facts $p \in \mathcal{F}$, and whenever they add a new fact $\mathit{head}(r)$ for ground rule r, they set

$$v(\mathit{head}(r)) = w(r) + \begin{cases} \sum_{Q \in \mathit{body}(r)} v(Q), & \text{if computing } h^{\text{add}} \\ \max_{Q \in \mathit{body}(r)} v(Q), & \text{if computing } h^{\max} \end{cases}$$

where $w(r)$ is the weight of the rule r. They then showed that if the goal is reachable, then $v(\texttt{goal}) = h^{\text{add}}(s)$ or $h^{\max}(s)$, depending on which one they are computing. This allows them to also early-stop the algorithm once they have computed $v(\mathit{goal})$.

Corrêa et al. (2022) further extended the above approach to compute the h^{FF} heuristic. Specifically, they attach an annotation to each rule in $\mathcal{R}$, where the annotations can be understood as a Domain Specific Language (DSL). They then compute $v(\texttt{goal})$ using the h^{add} formulation. Once reaching `goal`, they then backchain to compute the derivation of `goal`, and execute all annotations attached to the rules in the derivation. These annotations add the actions whose ground rule was used to reach `goal` to a set, and showed that $h^{\text{FF}}(s)$ is the sum of the costs of the actions in this set.

2.2.4 Do heuristics have to represent cost-to-goal?

So far, our discussion on heuristic functions have assumed the idea that heuristics functions should try to estimate the cost to reach the goal from a given state. As argued in Garrett, Kaelbling, and Lozano-Pérez (2016); Ferber et al. (2022); Chrestien et al. (2023); Hao et al. (2024); Chen and Thiébaux (2024), this is not necessarily the best target for satisficing planning, both theoretically and empirically. These works are mostly in the context of learning heuristic functions rather than constructing domain independent heuristic functions. Here we summarise some of their discussions on why they choose alternative targets for their learned heuristic functions, and leave a discussion of how they do it to Section 2.3 and Chapter 6.

In satisficing planning, planners generally employ the GBFS search algorithm since the goal is to solve the task quickly and plan quality is not crucial. A key property of GBFS is that the heuristic is only used to rank the search nodes. Two heuristic functions will lead to the exact search outcome as long as their relative ordering of search nodes is the same. Intuitively, it is therefore unnecessarily restrictive to expect heuristics to estimate the cost-to-goal.

On the other hand, for optimal planning where the A* search algorithm is typically used, cost-to-goal heuristics are also not necessarily the best. Heuristics that estimate cost-to-goal accurately or even perfectly (i.e., the perfect heuristic) will give the same value to many equally good states. This forces A* to have to explore a possibly exponential number of equally good solutions, as shown in Helmert and Röger (2008). The purpose of planning is typically to just finding one plan. It may therefore be more efficient to use an admissible heuristic that biases towards a certain optimal plan and is less accurate, rather than one that only estimates cost-to-goal.

Lastly, in the context of learning heuristic functions, choosing alternative targets allows using training information not otherwise available when choosing cost-to-goal as target. Specifically, given a training plan, using cost-to-goal as target only allows using states on the training plan trace, as cost-to-goal information is typically expensive for states not in the training plan trace. Using alternative targets, as done in Garrett, Kaelbling, and Lozano-Pérez (2016), Hao et al. (2024), and Chen and Thiébaux (2024), allows for using states off the training plan trace. This is particularly useful as states on the training plan trace only give indications of what is ideal, while states off the training plan trace indicate what might not be ideal.

2.3 Graphs and Planning

In this section we discuss graphs their use in planning, with the ultimate goal of explaining the state-of-the-art WL-GOOSE system for learning planning heuristics. Specifically, we briefly explore graph neural networks in Section 2.3.1. We then explain in more detail the Weisfeiler-Lehman kernel in Section 2.3.2 and its use in planning in Section 2.3.3. Lastly, we discuss the ranking versus regression approaches of training planning heuristics in 2.3.4. We leave a more general discussion of works in learning for planning, much of which use graphs, to Chapter 6.

2.3.1 Graph neural networks

Graph neural networks (GNNs) are a class of neural networks that operate on graph data structures. They have gathered significant attention due to the flexibility of graphs and the rise in popularity and success of neural networks due to developments in parallel computing and deep learning. GNNs have been used in both tangible applications such as social networks (Fan et al., 2019) and to solve abstract problems such as the NP-complete decision Travelling Salesperson Problem (Prates et al., 2019). For this section, we assume familiarity with neural networks and their training by optimising a differentiable loss function with backpropagation.

A graph is a tuple $G = \langle V, E \rangle$ where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. It is undirected if for each edge $(u, v) \in E$, (v, u) is also in E, and directed otherwise. In the context of GNNs, we often associate feature vectors with vertices and edges. GNNs learn to compute functions over graphs with these features, independently of the graph size and structure (Scarselli et al., 2009; Hamilton, 2020). GNNs operate in a message-passing framework, where each vertex aggregates and updates information from other vertices in the graph. For more complete overview we refer to Zhou et al. (2020) for a comprehensive survey of GNNs.

It is important to note that due to the message-passing nature, there are important limitations to GNNs. Specifically, it has been shown that GNNs cannot distinguish all pairs of graphs which are not isomorphic, but they can distinguish those distinguishable by the Weisfeiler-Lehman (WL) algorithm (Morris et al., 2019), which are in turn those distinguishable by the first-order logic with counting quantifiers and two-variables, C_2 (Barceló et al., 2020). We explore this in more detail in the next section.

2.3.2 The Weisfeiler-Lehman kernel

The Weisfeiler-Lehman (WL) algorithm is initially a heuristic test for answering the graph isomorphism problem.

Definition 5 (Graph isomorphism)**.** *The* graph isomorphism problem *is the problem of determining whether two graphs* $G_1 = \langle V_1, E_1 \rangle$ *and* $G_2 = \langle V_2, E_2 \rangle$ *are isomorphic, i.e., there exists a bijection* $\phi : V_1 \to V_2$ *such that* $(u, v) \in E_1$ *if and only if* $(\phi(u), \phi(v)) \in E_2$*.*

Algorithm 2: The WL algorithm

Data: A graph $G = \langle V, E \rangle$ and an injective hash function h which maps an integer and integer multiset pair to integers.

$c^{(0)}(v) \leftarrow 0, \ \forall v \in V$
for $i = 1, \ldots$ **do**
  $c^{(i)}(v) \leftarrow h\left(c^{(i-1)}(v), \left\{\!\!\left\{c^{(i-1)}(u) | u \in \mathcal{N}(v)\right\}\!\!\right\}\right), \ \forall v \in V$
  if $c^{(i)} = c^{(i-1)}$ **then**
    return $\left\{\!\!\left\{c^{(i)}(v) \mid v \in V\right\}\!\!\right\}$

The graph isomorphism problem is in NP, but it is not known to be in P or NP-complete, and hence usually given its own complexity class, GI. The WL algorithm is a simple and efficient algorithm for testing if two graphs are isomorphic. It is a one-sided test in that when it finds two graphs to be non-isomorphic, they are indeed non-isomorphic, but it cannot show that two graphs are isomorphic. It does so maintaining colours for each vertex, and iteratively updating the colours by hashing the colour of vertices and a multiset of the colours of its neighbours. Its pseudocode is given in Algorithm 2, where N is the function that maps vertices to their neighbours, i.e., $\mathcal{N}(v) = \{u \in V \mid (v, u) \in E\}$.

It is known that $|V| - 1$ is a tight bound for the number of iterations the WL algorithm needs to converge to a stable colouring (Kiefer and McKay, 2020). If the WL algorithm converges to the different multiset of colours for two graphs, then the graphs are not isomorphic. However, two non-isomorphic graph may converge to the same multiset of colours, and hence the WL algorithm cannot distinguish them. It has been shown that the WL algorithm is able to count substructures representable by first-order logic with counting quantifiers and two variables, C_2 (Cai, Fürer, and Immerman, 1992).

In addition to its use in graph isomorphism, the WL algorithm has been shown to be a powerful tool for transforming graphs into feature vectors that can be used in machine learning tasks. Specifically, the WL kernel is a kernel function that transforms graphs into vectors, whose values are the counts of each colour seen over a fixed number of iterations of the WL algorithm. These vectors are typically then used to compute the similarity between graphs, and have been shown to be effective in various machine learning tasks.

2.3.3 WL features for planning

The WL kernel has been very successfully applied to planning by (Chen, Trevizan, and Thiébaux, 2024). Specifically, their method involves three steps,

1. *Graph construction*: They convert planning states to graph representations. Their graph representations are undirected graphs where vertices are assigned colours and edges are assigned labels. These colours represent planning related information about the vertices and edges.

2. *WL kernel*: They extend the WL kernel to work with their graph representations. Their variation allows them to generate fixed length feature vectors for all graphs, where the vector length is determined from the training data.

3. *Learning*: They use the WL kernel feature vectors to train a classical machine learning model to predict the cost-to-goal of planning states. This produces a planning heuristic. We defer a general discussion of learning planning heuristics given feature vector representations of planning states to Section 2.3.4.

In more detail, Chen, Trevizan, and Thiébaux (2024) represent a planning state as its instance learning graph (ILG). The ILG includes vertices for each object in the planning task and for each atom in the state or goal. Vertices are coloured to indicate their nature, i.e., is the vertex an object, an atom in the state, an atom in the goal, or an atom in both. Vertices are connected by edges if the atom involves the object, and the edges are labelled by the position of that the object appears in the atom. The precise definition of the ILG is given in Definition 6.

Definition 6 (ILG)**.** *Given a planning task* $\Pi = \langle \langle \mathcal{P}, \mathcal{A} \rangle, \langle O, s_0, G \rangle \rangle$*, the* instance learning graph *(ILG) for a state* $s \in \mathbb{S}$ *is a graph* $G = \langle V, E, c, l \rangle$*, where,*

- $V = O \cup s \cup G$.
- $E = \bigcup_{p=P(o_1,\ldots,o_n)\in s\cup G} \{\langle p, o_1\rangle, \ldots, \langle p, o_n\rangle\}$.
- $c : V \to (\{\text{ap}, \text{ug}, \text{ag}\} \times \mathcal{P}) \cup \{\text{ob}\}$ *given by,*

$$c(v) = \begin{cases} \text{ob}, & \text{if } v \in O \\ (\text{ap}, P), & \text{if } v = P(o_1, \ldots, o_n) \in s \setminus G \\ (\text{ag}, P), & \text{if } v = P(o-1, \ldots, o_n) \in s \cap G \\ (\text{ug}, P), & \text{if } v = P(o-1, \ldots, o_n) \in G \setminus s \end{cases}.$$

- $l : E \to \{1, \ldots, n\}$*, where* n *is the maximum number of objects in any atom in* $s \cup G$*, given by,*

$$l(\langle P(o_1, \ldots, o_n), o_i\rangle) = i.$$

Once converting planning states to their ILGs, Chen, Trevizan, and Thiébaux (2024) then apply the WL kernel to these ILGs. They extended the WL kernel to account for the edge labels in their ILGs, by using the edge labels in the hashing function. The specific algorithm for this is given in Algorithm 3. Here, the resulting multisets of colours are used as feature vectors, where the value at index i in the vector is the count of the i-th colour in the multiset. Given a training dataset, the collection of all colours seen in the training set is used to determine the length of the feature vectors. These feature vectors are then used to train a machine learning model to predict the cost-to-goal of planning states.

Algorithm 3: The WL algorithm, extended to account for edge labels by Chen, Trevizan, and Thiébaux (2024)

Data: A graph $G = \langle V, E, c, l \rangle$ with vertex colours and edge labels, an injective hash function h which maps an integer and a multiset of integer pairs to integers, and an iteration limit L.

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

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

  $c^{(i)}(v) \leftarrow h\left(c^{(i-1)}(v), \{\!\{(c^{(i-1)}(u), l(\langle u, v \rangle)) | \langle u, v \rangle \in E\}\!\}\right),\ \forall v \in V$

return $\{\!\{c^{(i)}(v) \mid v \in V, 0 \leq i \leq L\}\!\}$

An issue with this approach is that colours not seen during training but seen at test time when using the learned heuristic will not be accounted for in the feature vectors. These so-called *unseen colours* can lead to poor generalisation of the learned heuristic. An additional issue is *colour explosion*, where the number of colours seen in the training set increases rapidly with the iteration limit L, resulting in large and sparse feature vectors. In an attempt to mitigate both issues, the authors proposed in a follow-up work (Chen and Thiébaux, 2024) to change to the multiset in the input to the WL hashing function to a set instead. This groups together different inputs that vary only in the count of their colours, and hence reduces the number of unseen colours and overall colours. Colour explosion is also dealt with by not including static atoms, i.e., atoms whose value cannot change, in the ILG.

2.3.4 Ranking versus regression

Once the feature vectors are computed, they are used to train a classical machine learning model, which represents a heuristic for a particular domain. The training dataset is a set of training tasks $\Pi_1, \ldots, \Pi_m$ in the same domain, and for each task a training plan $\pi_1, \ldots, \pi_m$. Each training plan is a sequence of actions $a_1, \ldots, a_n$ that solves the corresponding task by traversing the sequence of states $s_0, \ldots, s_n$, where s_0 is the initial state and s_n is a goal state. Given such training data, Chen, Trevizan, and Thiébaux (2024) initially trained heuristics through regression, and later trained heuristics through ranking in Chen and Thiébaux (2024).

Regression heuristic In the regression approach, the heuristic is trained by learning to map each state in the training plan to its cost-to-goal. Given a sequence of states $s_0, \ldots, s_n$ from a training plan, their respective regression targets are $n, n-1, \ldots, 0$. This results in the set of training data $\{(\phi(s_0), n), \ldots, (\phi(s_n), 0)\}$, where ϕ is the function that maps states to feature vectors using the WL kernel and ILG. The entire training dataset $\mathcal{D}$ is the collection of all such sets from all training plans.

Given the training dataset $\mathcal{D}$, the regression heuristic is then obtained by training a classical machine learning model to minimise the mean squared error on the training

set, i.e.,

$$\arg\min_{\theta} \frac{1}{|\mathcal{D}|} \sum_{(\phi(s),g)\in\mathcal{D}} \left(h_\theta(\phi(s)) - g\right)^2,$$

where h_θ is the machine learning model with parameters θ.

In theory, the machine learning model can be any regression model. In practice, Chen, Trevizan, and Thiébaux (2024) experimented with support vector regression (SVR) with the dot product kernel and radial basis kernel, as well as Gaussian process regression (GPR) with the dot product kernel. They found that GPR with the dot product kernel performed the best on the learning track of the International Planning Competition (IPC) 2023. Specifically, the regression heuristics they learned outperformed the well-known FF heuristic, and was able to get close in performance the state-of-the-art LAMA planner by itself, without the use of extra planning techniques that are employed by LAMA. Moreover, a major benefit of heuristics learned this way is the training efficiency. Chen, Trevizan, and Thiébaux (2024) reported that training these regression heuristics typically only require a few minutes on a single CPU core. Altogether, these results show that the WL kernel allows obtaining very powerful domain-specific heuristics very cheaply in a domain-independent fashion.

Ranking heuristic As we had discussed in Section 2.2.4, cost-to-goal is not necessarily the best target for learning planning heuristics. Various works have argued for a ranking approach over the regression approach (Garrett, Kaelbling, and Lozano-Pérez, 2016; Ferber et al., 2022; Chrestien et al., 2023; Hao et al., 2024). In the ranking approach, the heuristic is trained by learning to rank states in the training plan better than their predecessors and states off the training plan.

Various ways of generating training datasets for ranking heuristics using plan traces have been proposed. Here we discuss the approach proposed in (Hao et al., 2024), which is used in the WL-GOOSE system (Chen and Thiébaux, 2024). This approach was shown to generate smaller training datasets that resulted in better generalisation of the learned ranking heuristics.

Given a training plan $\pi = a_1, \ldots, a_n$ and the corresponding sequence of states $s_0, \ldots, s_n$, the training dataset for the ranking heuristic consists of tuples of the form $\langle \mathbf{x}, \mathbf{x}', \delta \rangle$. Each such tuple indicates that the feature vector $\mathbf{x}$ should be given a value δ lower than that given to the feature vector $\mathbf{x}'$. The preference for lower values comes from the fact that planning heuristics are expected to give lower values to better states. Two types of tuples are generated from the state sequence using the function ϕ that transforms states to feature vectors, specifically,

1. *Predecessors*: Each state s_i with $i > 0$ is ranked better than its immediate predecessor, with the gap δ being the cost of the action applied in its predecessor in the training plan. Specifically, for each $i > 0$, the tuple $\langle \phi(s_i), \phi(s_{i-1}), c(a_i) \rangle$ is generated.

2. *Siblings*: Each state s_i with $i > 0$ is ranked better than all states s'_i, where s'_i is a successor state of s_{i-1} with $s_i \neq s'_i$. Here, the training plan only indicates that s_i is better than s'_i, but not by how much, so the gap δ is set to 0. Specifically, for each $i > 0$, the tuple $\langle \phi(s_i), \phi(s'_i), 0 \rangle$ is generated for all s'_i.

The complete dataset $\mathcal{D} = \{\langle \mathbf{x}_i, \mathbf{x}'_i, \delta_i \rangle \mid 1 \leq i \leq |\mathcal{D}|\}$ is the collection of all such tuples from all training plans. Given such a dataset, Chen and Thiébaux (2024) obtain a ranking heuristic by training a variation of a L1-regularised Rank Support Vector Machine (RankSVM), which is equivalent to solving the following mixed integer linear program (MILP),

$$\begin{aligned} \min_{\mathbf{w},\mathbf{z}} \quad & C \sum \mathbf{z}_i + \|\mathbf{w}\|_1 & \\ \text{s.t.} \quad & \mathbf{z}_i \geq 0 & \forall i \\ & \mathbf{w}^T(\mathbf{x}_i - \mathbf{x'}_i) \geq \delta_i - \mathbf{z}_i & \forall i \\ & \mathbf{w}_i \in \{-1, 0, 1\} & \forall i \end{aligned}$$

Here, $\mathbf{w}$ is the weight vector of the RankSVM, $\mathbf{z}$ is a vector of slack variables, and C is a positive regularisation hyperparameter, where a higher value results in weaker regularisation. The learned ranking heuristic is then given by the function $h_{\mathbf{w}}(s) = \mathbf{w}^T \phi(s)$.

To our knowledge, no direct comparison between the regression and ranking approaches using WL features has been published, such comparisons have been made for numeric planning. Numeric planning is an extension of classical planning with numeric state variables that appear in action preconditions and effects. Chen and Thiébaux (2024) extended their WL-GOOSE system to numeric planning, and showed that both regression and ranking heuristics learned with WL features outperformed existing state-of-the-art numeric planners. Furthermore, they found that the ranking heuristic outperformed the regression heuristic, which matches the expectation from various literature on learning heuristics for planning.

Chapter 3

A Natural Hierarchy of Action Sets

The planning task, or more generally the task of decision-making, is inherently hierarchical. Here we focus on the hierarchical nature of a single action. For example, when you decide you would like to go out to eat, you may first decide what type of food you would like to eat, then decide which particular restaurant, and even which dish you would like to order. Only once you have decided all these, have you actually completed making the decision to go out to eat. This hierarchy of decisions is not only natural but also efficient — making a decision at a higher level of abstraction allows ignoring the details until they are needed. Continuing with the example, once deciding not to eat a particular cuisine, there is no need to consider any related restaurants or dishes.

Current dominant approaches to classical planning mostly treat individual actions as entirely independent entities. This is akin to considering all combinations of cuisines, restaurants, and dishes at the same time. Such a process is deeply in contrast to the way humans typically make decisions. On a more grounded note, this ignores the way actions are constructed from action schemas in the Planning Domain Definition Language (PDDL) (Haslum et al., 2019), which as we show soon, induces natural relations on actions. Using it in Section 3.1, we construct a hierarchy of *partial actions*, which represent sets of related actions. We show in Section 3.2 how this hierarchy can be used to guide the search process in a planning system through a novel search space called *partial space search*. Then in Section 3.3, we also explain how partial space search allows for a more focused and efficient search. Our motivation for partial space search is to design a more learning-friendly search process, which we discuss in Section 3.4.

Although our focus is on classical planning, it is important to note that our contributions in this chapter can be easily extended to more expressive forms of planning. In theory, any form of planning where actions are instantiated from an action schema using objects can benefit from our approach with minimal modifications.

3.1 Partial Actions

The way that humans naturally make decisions is often hierarchical. Such a process, as we discussed in the example earlier, involves making gradual, small step decisions that eventually lead to a final decision. The way PDDL models action schemas naturally models this process. As we explained in Section 2.1, an action schema is a template for an action with a set of parameters. For example, the action schema for an action that moves a block may have parameters for the block to be moved, the block's source, and the block's destination. An action is then instantiated from an action schema by replacing each parameter with an appropriate object. This instantiation process induces a natural hierarchy on sets of actions, which we discuss next using *partial actions*.

A partial action models an intermediate step in the process of instantiating an action from an action schema, where some parameters have been instantiated. At the same time, we also view partial actions as a set of related actions that agree on the parameters that have been instantiated. Formally, we define a partial action as follows:

Definition 7 (Partial action)**.** *A* partial action *is an action schema with some (including none and all) of its parameters instantiated. We typically denote a partial action in the form $A(o, \_)$, where A is the action schema, o is the object that instantiates the first parameter, and "$\_$" denotes an uninstantiated parameter. Where clear from context, we also directly denote the same partial action as $A(o)$. Furthermore, we use the special partial action* None *to denote the partial action where not even the action schema has been chosen.*

For this work, we will assume there is a fixed order of parameters in an action schema and that partial actions have a prefix of the ordered parameters instantiated. In practice, we use the order in which the parameters are defined in the PDDL representation. In theory, any fixed order can be used, and different orders induce different hierarchies, some of which may be more appropriate. For example, it likely makes more sense to first decide cuisine then restaurant, rather than the other way around. We leave the exploration of different orders for future work.

The fixed order of parameter instantiation results in the formation of a tree of partial actions for any given planning domain task, as defined below:

Definition 8 (Partial action tree)**.** *The* partial action tree *of a lifted planning task is a tree where each node is a partial action, and the root node is* None*. The children of* None *are the partial actions that are action schemas with no parameters instantiated. The children of any other partial action $A(o_1, \ldots, o_i, \_, \ldots, \_)$ are the partial actions of the form $A(o_1, \ldots, o_i, o_{i+1}, \_, \ldots, \_)$ for all possible objects o_{i+1} that can instantiate the next parameter. The leaves of the tree are the partial actions that are fully instantiated actions.*

An example of a partial action tree is shown in Figure 3.1. To help refer to partial actions at different levels of the tree, we introduce the notion of *specificity*. The specificity of a

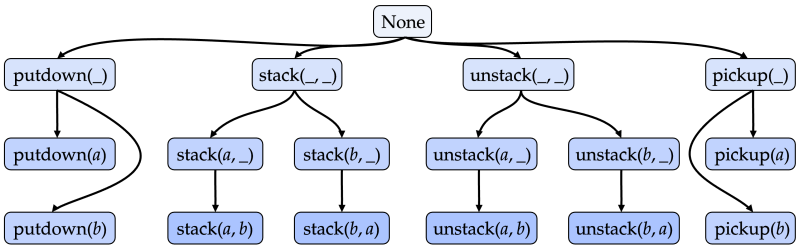


Figure 3.1: Example of a partial action tree for a Blocksworld task with objects a and b. The node colours indicate specificity, with darker colours indicating higher specificity.

node is its depth in the partial action tree, e.g., the specificity of None is $\text{spec}(\text{None}) = 0$, and the specificity of a partial action with k parameters instantiated is $k + 1$.

As we mentioned, a partial action ρ is a representation of the ground actions that agree on the action schema and the parameters that have been instantiated. This is equivalent to saying that ρ represents the leaf (ground actions) of the partial action tree in its subtree. We use $\mathbb{A}^\rho$ to denote this set of ground actions. For example, in Figure 3.1, the partial action putdown(_) represents the set of ground actions $\mathbb{A}^{\text{putdown(\_)}} = \{\text{putdown}(a), \text{putdown}(b)\}$. Recall that we use $\mathbb{A}_s$ to denote the set of applicable actions in a state s — we will additionally use $\mathbb{A}_s^\rho$ to denote the intersection of $\mathbb{A}_s$ with $\mathbb{A}^\rho$, i.e., the set of ground actions represented by the partial action ρ that are applicable in state s. Furthermore, we say that a partial action ρ is *applicable* in a state s if $\mathbb{A}_s^\rho \neq \emptyset$. Given these definitions, we can introduce some basic properties of partial actions:

Proposition 1. *Given a lifted planning task, there are finitely many partial actions and the partial action tree is finite.*

Proof. Since the number of objects is finite and the number of parameters in an action schema is finite, there are finitely many ways to instantiate a parameter. Therefore, there are finitely many partial actions. Since the number of partial actions is finite, the partial action tree is also finite. □

Proposition 2. *For any partial action ρ and ρ' such that ρ is an ancestor of ρ' in the partial action tree, $\mathbb{A}^{\rho'} \subseteq \mathbb{A}^\rho$ and $\mathbb{A}_s^{\rho'} \subseteq \mathbb{A}_s^\rho$ for all states s.*

Proof. Since $\mathbb{A}^\rho$ and $\mathbb{A}^{\rho'}$ are the sets of leaves in the subtree of ρ and ρ' respectively, the first part of the proposition is immediate. The second part follows from the first part and the definition of $\mathbb{A}_s^\rho$ and $\mathbb{A}_s^{\rho'}$ being the intersection of $\mathbb{A}_s$ with $\mathbb{A}^\rho$ and $\mathbb{A}^{\rho'}$ respectively. □

Proposition 3. *For any partial action ρ with children $\rho_1, \ldots, \rho_n$, the set $\mathbb{A}^\rho$ is the union of the sets $\mathbb{A}^{\rho_1}, \ldots, \mathbb{A}^{\rho_n}$. Furthermore, for any state s, the set $\mathbb{A}_s^\rho$ is the union of the sets*

$\mathbb{A}_s^{\rho_1}, \ldots, \mathbb{A}_s^{\rho_n}$. *Lastly,* ρ *is applicable in state* s *if and only if at least one of its children is applicable in state* s.

Proof. The first part of the proposition follows from the definition of $\mathbb{A}^\rho$ as the leaves in the subtree of ρ. The second part follows from the first part and the definition of $\mathbb{A}_s^\rho$ as the intersection of $\mathbb{A}_s$ with $\mathbb{A}^\rho$. The last part follows from the second part and the definition of the applicability of a partial action in a state. □

The above propositions show that partial actions form a natural hierarchy of sets of actions through the partial action tree. This hierarchy can be used to allow for refining from the more abstract to the more concrete. Specifically, we can use it to refine the set of applicable actions in a state, i.e., those represented by None, to more specific sets of actions, by simply walking down the partial action tree to applicable children. This is the basis of our novel approach to search in planning, which we discuss in the next section.

3.2 Partial Space Search

In traditional state space search, search nodes represent states, and the search process simply explores the state space by expanding states with their successor states. State space search is, strictly speaking, a translation of the planning task into a search problem, whose solution is plan. Given a task Π, we will refer to the state space search problem as $\mathrm{S}^3(\Pi)$.

In state space search, each successor state is generated by applying an applicable action to the current state. This is how state space search ignores the hierarchical nature of actions, as it treats all actions as independent entities.

Through partial actions, we have introduced a hierarchy of action sets in the form of a partial action tree. Using it, we can guide the search process to gradually refine the set of applicable actions in a state by walking down the tree. This is exactly what we do in our novel search problem formulation called *partial space search*:

Definition 9 (Partial space search)**.** *Given a lifted planning task* Π*, the* partial space search *of* Π *is a search problem* $\mathrm{PS}^2(\Pi)$ *where search nodes are pairs of the form* $\langle s, \rho \rangle$*, where* s *is a state in* $\mathbb{S}$ *and* ρ *is a partial action. The search process starts at the root node* $\langle s_0, \mathrm{None} \rangle$*, where* s_0 *is the initial state of* Π*. The successor nodes of each node* $\langle s, \rho \rangle$ *is given by*

- *If* ρ *is not fully instantiated, then the successor nodes are* $\langle s, \rho' \rangle$ *for all partial actions* ρ' *that are children of* ρ *in the partial action tree and applicable in state* s*.*
- *If* ρ *is fully instantiated, then the successor nodes are* $\langle s', \mathrm{None} \rangle$*, where* s' *is the state resulting from applying the ground action represented by* ρ *to state* s*.*

A goal node of $\mathrm{PS}^2(\Pi)$ *is a node of the form* $\langle s, \mathrm{None}\rangle$ *where* s *is a goal state of* Π. *The resulting plan from reaching such a goal node is the sequence of ground actions represented by fully instantiated partial actions along the path from the root node to the goal node.*

Given the partial space search of a planning task, we seek to use it to solve the planning task by finding a path from the root node to a goal node using a search algorithm. The search algorithm can be any standard search algorithm, such as A* search or greedy best-first search (GBFS). The key difference is that any heuristic used in the search algorithm should be able to evaluate the state and partial action pairs, rather than just states in traditional state space heuristic search. We discuss this in detail in Chapter 4.

Example 1. *Consider the Blocksworld task with objects* a, b, *and* c. *Suppose we are in the state* s_0 *where* a *is stacked on top of* b, *and blocks* b *and* c *are on the table. The goal is to have all three blocks on the table. Using partial space search, we would have the initial search node* $n_0 = \langle s_0, \mathrm{None}\rangle$.

1. *The successor nodes of* n_0 *are* $n_1 = \langle s_0, \mathit{unstack}(\_, \_)\rangle$ *and* $n_1' = \langle s_0, \mathit{pickup}(\_)\rangle$, *representing the two applicable action schemas in* s_0. *Suppose we choose to expand* n_1, *based on guidance from a heuristic.*

2. *The only successor node of* n_1 *is* $n_2 = \langle s_0, \mathit{unstack}(a, \_)\rangle$, *since* a *is the only block that can be unstacked in* s_0. *We will expand* n_2.

3. *Again,* n_2 *only has one successor, being* $n_3 = \langle s_0, \mathit{unstack}(a, b)\rangle$. *We will expand* n_3.

4. *Since the partial action in* n_3 *is fully-instantiated, we will generate the successor state* s_1 *of applying the ground action* $\mathit{unstack}(a, b)$ *to* s_0. *The resulting state* s_1 *is where* a *is held, with* b *and* c *on the table. We will then generate the successor node* $n_4 = \langle s_1, \mathrm{None}\rangle$.

5. *The successor nodes of* n_4 *are* $n_5 = \langle s_1, \mathit{putdown}(\_)\rangle$ *and* $n_5' = \langle s_1, \mathit{stack}(\_, \_)\rangle$, *representing the two applicable action schemas in* s_1. *Suppose we choose to expand* n_5.

6. *The only successor node of* n_5 *is* $n_6 = \langle s_1, \mathit{putdown}(a)\rangle$, *since* a *is the only block that can be put down in* s_1. *We will expand* n_6.

7. *Since the partial action in* n_6 *is fully-instantiated, we will generate the successor state* s_2 *of applying the ground action* $\mathit{putdown}(a)$ *to* s_1. *This generates the successor node* $n_7 = \langle s_2, \mathrm{None}\rangle$. *We will expand* n_7.

8. n_7 *is a goal node, as* s_2 *is a goal state. The search terminates with the found plan* $\mathit{unstack}(a, b), \mathit{putdown}(a)$.

Below we show soundness and completeness for partial space search, and how it relates to state space search.

Lemma 1. *Given a lifted planning task* Π*, the set of plans that can be found by the partial space search of* Π *is equal to the set of plans that can be found by the state space search of* Π*.*

Proof. We first show that the set of plans found by partial space search is a subset of those found by state space search. Let $\pi = a_1, \ldots, a_n$ be a plan found by the state space search of Π where each a_i is a ground action. Let $s_0, \ldots, s_n$ be the sequence of states where s_0 is the initial state of Π and s_i is the state resulting from applying a_i to s_{i-1} for $i > 0$. We show that there is a path from the root node $\langle s_0, \text{None} \rangle$ to the goal node $\langle s_n, \text{None} \rangle$ in the partial space search of Π that is equivalent to π.

Specifically, let k_i be the number of parameters of the action schema of the action a_i. For each i, let ρ_i^j be the partial action with specificity j such that $\mathbb{A}^{\rho_i^j}$ contains a_i, where j ranges from 0 to $1 + k_i$. Such a ρ_i^j is unique as the partial action tree is a tree, and all partial actions with specificity j are at the same depth, so only one can be an ancestor of a_i. Furthermore, ρ_i^j is applicable in state s_{i-1} as a_i is applicable in state s_{i-1}. Moreover, $\langle s_{i-1}, \rho_i^j \rangle$ is a successor node of $\langle s_{i-1}, \rho_{i-1}^{j-1} \rangle$ in the partial space search of Π by definition of partial space search. Lastly, $\langle s_{i-1}, \rho_i^{1+k_i} \rangle$ is the predecessor node of $\langle s_i, \text{None} \rangle$. Therefore, the sequence of nodes $\langle s_0, \text{None} \rangle, \langle s_0, \rho_1^1 \rangle, \ldots, \langle s_{n-1}, \rho_n^{1+k_n} \rangle, \langle s_n, \text{None} \rangle$ is a path in the partial space search of Π, and the sequence of ground actions $\rho_1^{1+k_1}, \ldots, \rho_n^{1+k_n}$ equals π.

The reverse direction of showing that the set of plans found by the partial space search is a superset of those found by the state space search is even easier. Suppose π is a plan found by the partial space search of Π. Then the sequence of ground actions represented by the fully instantiated partial actions along the path from the root node to the goal node is clearly a plan that can be found by the state space search of Π. □

Theorem 1. *Partial space search is sound and complete for solving planning tasks.*

Proof. Since the state space search of a planning task is sound and complete, so is the partial space search of the same planning task by the above lemma. □

In practice, the partial space search problem $\text{PS}^2(\Pi)$ of many tasks often involve many expansions where there is only one successor node. To save need for unnecessary operations and heuristic evaluations, we repeatedly expand such nodes until obtaining multiple successor nodes. For instance, in Example 1, n_1 would expand directly into n_5 and n_5', skipping n_2, n_3, and n_4.

3.3 Efficiency of Partial Space Search

Partial space search aims to perform a more focused search by exploiting the hierarchy of partial actions. For instance, if a particular action schema is not useful in a given state, it is unnecessary to evaluate each of its instantiations individually, as required in state space search. Partial space search eliminates the need for such individual expansions—if

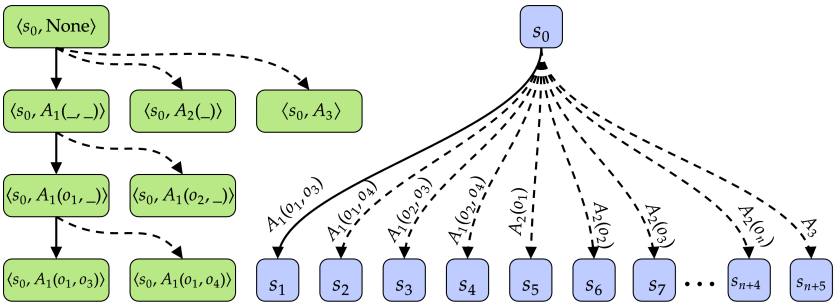


Figure 3.2: Example of the expansion tree of a state s_0 for partial space search (left) and state space search (right). Dashed lines indicating possible successor nodes that require evaluations, while solid lines indicate expanded nodes. The decisions on which state to expand are made using an informed heuristic.

the action schema is irrelevant, the corresponding partial space search node will not be expanded.

Consider the example in Figure 3.2. By employing partial space search, the search process determines that action schemas A_2 and A_3 are unnecessary, thus avoiding their expansion. This allows it to bypass evaluating all the actions produced by the instantiations of A_2 and A_3. Compared to state space search starting from the same node, this represents a significant reduction in the number of evaluations required. This is the key advantage of partial space search over state space search—offering a more efficient and focused search process.

However, the efficiency of partial space search comes with trade-offs. As shown in Figure 3.2, partial space search may need to expand multiple nodes to determine the correct action in state s_0, while state space search only requires expanding one. This is a defining property of partial space search: it transforms the state space tree by reducing its branching factor while increasing its depth, breaking down each original search step into multiple smaller ones.

The efficiency of partial space search depends on how well the planning task's actions can be broken down. For example, consider an action schema with three parameters, where there are n_1, n_2, and n_3 choices for each parameter. State space search would require evaluating all $n_1 n_2 n_3$ possible actions, while partial space search—guided by a heuristic—may only need to evaluate $n_1 + n_2 + n_3$ partial actions. In cases where n_1 and n_2 are small (e.g., both equal to one), partial space search would be less efficient, as the actions do not factor well. However, even in such scenarios, the additional cost is limited to evaluating only a few more nodes. On the other hand, when actions are nicely factored, the potential gains are substantial.

To further illustrate this, consider the toy domain VisitSome, where a robot must visit specific locations in an unbounded n-dimensional grid. When the robot is at location

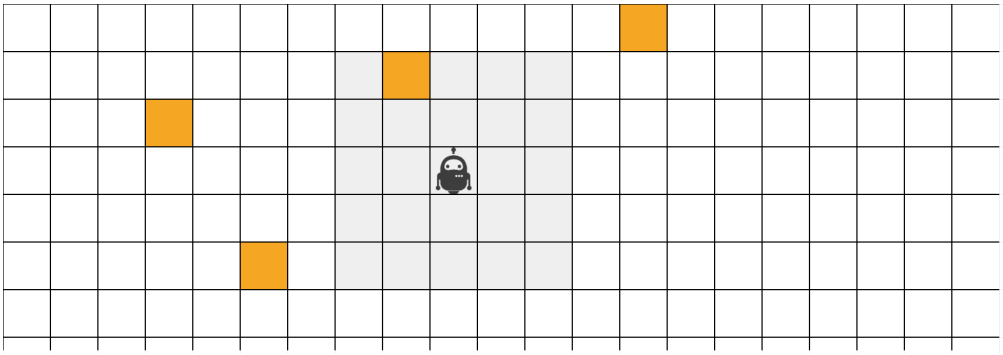


Figure 3.3: The VisitSome toy domain, see text for a detailed description of the domain. Here we show a 2-dimensional example where the robot can move in one action to any location within distance 2 (under the uniform norm L_∞) of its current location (indicated by grey shading). The orange shading indicates the goal locations the robot must visit.

$l = (x_1, \ldots, x_n)$, it can move in a single action to any location $l' = (y_1, \ldots, y_n)$ where $x_i - k \leq y_i \leq x_i + k$ for all i, with some fixed k. The only action schema in this domain has $2n$ parameters: the first n parameters determine l, and the last n determine l'. In state space search, the number of applicable actions in any state is $(2k + 1)^n$, as this represents the number of possible locations the robot can move to[1]. In partial space search, with an informed heuristic, the branching factor for the first n parameters is 1, while for the last n parameters it is $2k+1$. Thus, the total branching factor and number of evaluations needed is $n(2k + 1)$. For even small values of n and k, this can result in a significant reduction. For example, with $n = 3$ and $k = 2$, the branching factor reduces from 125 to 15.

In summary, partial space search enables a more directed and efficient search process by avoiding the evaluation of many irrelevant actions early on. The extent of efficiency gains depends on the structure of the planning task, but even in the worst case, the additional cost is minimal. Conversely, when actions are well-factored, the potential gains can be significant.

3.4 Why Does This Matter for Learning?

So far, our discussion of partial space search has focused solely on the search process, setting aside the main motivation for this work: designing a more learning-friendly planning component. In this section, we explain why partial space search is exactly what we had set out to achieve. Specifically, we argue that partial space search is a learning-friendly search process that leverages the learning component more and provides more

[1] Note that we count the robot's current location as one of the possible locations, allowing it to stay in one place indefinitely.

for the learning component to learn from.

Partial space search decomposes the search process into smaller steps, where well-informed heuristics can eliminate poor choices early. While this benefits both learning and non-learning heuristics, learning heuristics benefit more for several reasons:

1. **Informedness versus speed trade-off:** Search heuristics must balance between being informative and being computationally efficient. Partial space search increases the potential reward for being informative, allowing such heuristics to more effectively guide the search process. This shifts the trade-off between speed and accuracy towards the latter, benefiting learning heuristics in particular. Recent advances in machine learning, particularly deep learning, have led to increasingly powerful and computationally expensive models. Partial space search accommodates these slower but more accurate heuristics, making better use of the learning component.

2. **Easier adaptation:** Traditional state space heuristics are often designed specifically for state space search, as we had discussed in Section 2.2.3. Despite our proposed methods in Chapter 4 for automatic translation, such translated heuristics are not truly designed for partial space search. In contrast, methods for learning heuristics are generally more flexible. In theory, the methods discussed in Section 2.3 can be applied to learn heuristics for any search space as long as there is an appropriate graph representation of search nodes. This flexibility allows learning heuristics to adapt more easily to partial space search compared to traditional heuristics.

3. **More training data:** Chapter 4 discusses the methods we use to generate training data for learning heuristics for partial space search. By breaking the search process into smaller steps, partial space search generates more training data from the same training inputs when compared to state space search. Although more training data is not inherently better, and its quality is difficult to quantify, we show in our empirical evaluations that the additional training data provided by partial space search is beneficial for learning better heuristics. Specifically, heuristics learned from datasets generated by partial space search outperform those learned from datasets generated by state space search, even if both are used with state space search. In other words, partial space search provides more for the learning component to learn from.

In conclusion, partial space search can improve the efficiency of the search process and enhances the effectiveness of the learning component. By supporting slower and more accurate heuristics and providing more training data, partial space search is a learning-friendly search space that allows for more effective decision-making.

Chapter 4

Action Set Heuristics

To make partial space search truly effective, having strong heuristics functions is a necessity. As introduced in Section 2.2, heuristic functions map search nodes to real numbers that estimate the quality of search nodes. There, we had focused on state space heuristics where search nodes represent planning states. Partial space search, however, requires heuristics that can evaluate search nodes that are state and partial action pairs. In this chapter, we define and discuss how to obtain such heuristics. Specifically, in Section 4.1, we define action set heuristics — a more general form of the heuristics we need for partial space search, which can evaluate search nodes that are state and action set pairs. These heuristics are suitable for partial space search as given a state, partial actions represent sets of applicable actions on that state. In Section 4.2, we explain how to automatically translate any existing state space heuristic to an action set heuristic and the drawbacks associated with this translation. In particular, we discuss how to efficiently translate the h^{FF} heuristic. In Section 4.3, we discuss how to represent state and action set pairs as graphs, which be used to generate feature vectors through graph kernels, as introduced in Section 2.3.3. Lastly, given these feature vectors, we discuss how to train action set heuristics in Section 4.4.

4.1 Definition

Our goal in this section is to define heuristics to guide partial space search. Given a state s and a partial action ρ, we had discussed in Section 3.1 that ρ is a representation of the set of ground and applicable actions $\mathbb{A}_s^\rho$. This representation helps simplify our discussion and motivates our definition of action set heuristics.

Definition 10 (Action set heuristic)**.** *An* action set heuristic *is a function* $h : \mathbb{S} \times 2^{\mathbb{A}} \to \mathbb{R} \cup \{\infty\}$*. Given a state* s *and a set of actions* Λ*,* $h(s, \Lambda)$ *estimates the quality of the actions* Λ *when applied in the state* s*. It is generally expected, although not strictly*

required that:

- *any inapplicable action in* Λ *is ignored, i.e.,* $h(s, \Lambda) = h(s, \Lambda \cap \mathbb{A}_s)$*;*
- *if* s *is a dead-end state, or if all the actions in* Λ *are inapplicable in* s*, then* $h(s, \Lambda) = \infty$*.*

As a shorthand, for any partial action ρ*, we denote* $h(s, \mathbb{A}_s^\rho)$ *as* $h(s, \rho)$*.*

The definition of action set heuristics is intentionally general. It allows action set heuristics to be used for any search space that consider set of actions like partial space search. To our knowledge, partial space search is the only such search space. However, we hope future research will explore other search spaces that can benefit from action set heuristics.

The above definition is in fact so general that we have made no comment on what it really means to estimate the quality of a set of actions. Different interpretations of what this means will lead us to obtaining different action set heuristics, as we will explore in the rest of this chapter. However, it is worth discussing here a particular class of action set heuristics. Specifically, given a state space heuristic h and a set of actions Λ, we can consider the values of h on the states reachable by applying the actions in Λ in the state s. This way, we can define an action set heuristic h' that estimates the quality of a set of actions Λ in a state s as an aggregation of these values from h. This simple translation from state space heuristics to action set heuristics defeats the purpose of action set heuristics. Our goal here is to evaluate the set of actions at once — evaluating them one by one means we may as well go back to state space search.

It is clear that action set heuristics can be used to guide heuristic search algorithm in partial space search. It is important to note that action set heuristics can also be used to guide state space search. Specifically, given a state s and an action set heuristic h, we can obtain a state space heuristic h' by defining $h'(s) = h(s, \mathbb{A}_s) = h(s, \text{None})$. Here None is the special partial action at the root of the partial action tree we had defined in Section 3.1. This is a useful property, allowing us to evaluate the quality of action set heuristics both in state space search and partial space search. This way, we can then evaluate the impact of action set heuristics with that of partial space search in isolation.

4.2 Automatic Translation from State Space Heuristics

As we previously mentioned, naive translation of state space heuristics to action set heuristics is not efficient. However, we can still automatically obtain efficient action set heuristics from state space heuristics. We do so by viewing the heuristic value of a state and action set pair as the quality of the state with the immediately applicable actions restricted to the action set. This gives us the family of *restriction heuristics*.

Definition 11 (Restriction heuristic)**.** *For a lifted planning task* $\Pi = \langle\langle \mathcal{P}, \mathcal{A}\rangle, \langle O, s_0, G\rangle\rangle$*, given a state space heuristic* h*, a state* s*, and a set of actions* Λ*, the* Λ-restricted task

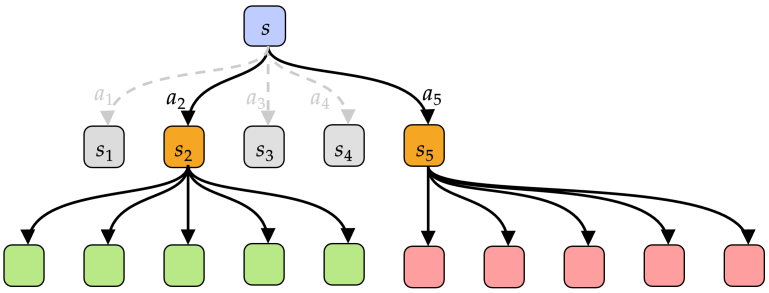


Figure 4.1: Illustration of the task transformation for the restriction heuristic. Here, we seek to compute the restriction heuristic $h_{\text{rs}}(s, \{a_2, a_5\})$. See text for more detail.

Π_Λ *is* Π *with the modifications:*

1. *an additional predicate* ϵ*, which takes no parameters and is false in the initial state, is added to the set of predicates* $\mathcal{P}$*;*
2. *for each action schema* $A \in \mathcal{A}$*, the extra precondition* ϵ *is added;*
3. *for each ground action* $a \in \Lambda$*, the additional add effect* ϵ *is added. In terms of the lifted task, this means adding fully instantiated action schemas corresponding to these ground actions to the set of action schemas* $\mathcal{A}$*. These added fully instantiated action schemas have* ϵ *not as a precondition, but as an add effect.*

Then, the restriction heuristic h_{rs} *is the action set heuristic defined by* $h_{\text{rs}}(s, \Lambda)$ *taking the value of* $h(s)$ *on the task* Π_Λ*. We say that* h_{rs} *is the* restriction *of* h*.*

The task transformation for the restriction heuristic is illustrated in Figure 4.1. The task transformation here forbids the immediate application of actions a_1, a_3, and a_4, as indicated by their colour. However, restriction heuristics do not restrict action applicability beyond the immediate actions, as indicated by the lack of greying out after a_2 or a_5 has been applied. The heuristic value $h_{\text{rs}}(s, \{a_2, a_5\})$ is then the value of $h(s)$ on the transformed task.

4.2.1 Efficient computation of the restriction of the FF heuristic

Restriction heuristics are a relatively straightforward way to automatically translate any state space heuristic to an action set heuristic. However, there are potential challenges with this translation, in particular with respect to its efficient computation. Many interesting state space heuristics require some form of preprocessing of the planning task. For example, we had seen in Section 2.2.3 that the lifted computation of the FF heuristic requires preprocessing of the planning task into a Datalog program and for transformations to be applied to this program to make the computation efficient. The restriction heuristic brings challenges here. Specifically, the planning task that the

translated state space heuristic h is applied to is dependent on the action set Λ. This means that, naively, the preprocessing step must be done for each action set Λ that we wish to evaluate. Given the potential large number of actions and hence action sets, preprocessing for each possible action set is clearly infeasible.

We can overcome this challenge by noting that in Definition 11, only the third modification to the planning task is dependent on the action set Λ. This modification only involves adding ground actions for each action in Λ. This means that we can potentially preprocess the task Π with the first two modifications, and then at evaluation time consider the third modification. How exactly this can be done is dependent on the particular state space heuristic. Here, we discuss how to efficiently compute the restriction heuristic for the h^{FF} heuristic.

Since our goal is to work with large and challenging planning tasks, we focus on the lifted computation of the restriction of the FF heuristic, based on the lifted computation introduced in Section 2.2.3 from Corrêa et al. (2022). This way, our method can be applied to any planning task, rather than just those that can be grounded.

Specifically, to compute the restriction $h^{\text{FF}}_{\text{rs}}$ of the FF heuristic, we first perform preprocessing of the planning task Π with the first two modifications as described in Definition 11. This yields a Datalog program. Then, given a state s and an action set Λ, we add temporary ground rules to the Datalog program for each action in Λ, corresponding to the third modification. For each such ground action, we add a rule that adds the ϵ predicate to the state when the action is applied. We then run the Datalog program on the state s, and obtain the value of the FF heuristic on the transformed task. This way, we can efficiently compute the restriction of the FF heuristic for any state and action set pair.

4.2.2 Drawbacks of restriction heuristics

Restriction heuristics provide an automatic way to obtain action set heuristics from state space heuristics. In terms of our overall goal of obtaining strong action set heuristics, they provide a lower bound on the quality of action set heuristics that we can obtain. However, there are some drawbacks to restriction heuristics, which motivate the need for more sophisticated methods to obtain action set heuristics, both learning and non-learning based.

An ideal property for an action set heuristic to have is that it should reward refining a set of actions down to just the good actions. Restriction heuristics do not have this property. Specifically, given a state space heuristic h, a state s, and two set of actions Λ_1 and Λ_2 such that $\Lambda_1 \subset \Lambda_2$, the task Π_{Λ_1} is a harder version of the task Π_{Λ_2}. This is because the former task forbids more actions than the latter. Consequently, depending on the nature of h, we will likely have $h_{\text{rs}}(s, \Lambda_1) \geq h_{\text{rs}}(s, \Lambda_2)$. That is, restriction heuristics generally do not reward action set refinement.

This is not something that cannot be overcome. Specifically, it is possible to reduce the

cost of actions in Λ, with the discount being stronger for smaller action sets than larger ones. This way, we can reward action set refinement for action-cost aware heuristics. In practice, we find that this did not benefit $h^{\text{FF}}_{\text{rs}}$ empirically in very limited experiments. We leave further investigation as future work.

Moreover, the generality of restriction heuristics means that they also ignore the entire point of action set heuristics — the action set. The task transformation simply compiles them away. This means that restriction heuristics are not able to capture that often, there is structure to the action set. For example, when action set heuristics are used to guide partial space search, the action set is induced from a partial action, which means that the actions in the action set are related in some way. Restriction heuristics do not capture this structure. This motivates the need for action set heuristics that are truly aware of the action set.

4.3 Graph Representations

Given our mission of developing a strong and integrated learning for planning system, and the drawbacks discussed in Section 4.2.2, it is only natural that we turn to learning-based methods to obtain action set heuristics. In this section, we introduce two novel graph representations for state and action set pairs. These two graph representations reflect different views on what the action set means. As we had seen in Section 2.3.3, we can immediately map these graph representations to feature vectors using the WL kernel with method introduced in Chen, Trevizan, and Thiébaux (2024). We will discuss how to use these graph representations to learn action set heuristics in Section 4.4.

Both of our graph representations are based on the Instance Learning Graph (ILG) from Chen, Trevizan, and Thiébaux (2024). As such, they share similar properties as the ILG. Specifically, they are undirected graphs where nodes and edges are coloured. Feature vectors are obtained from these graphs by running the Weisfeiler-Lehman kernel and collecting the count of each colour in the graph as these colours are updated over the iterations of the WL algorithm. Only the colours seen during training time are used to generate feature vectors, those not seen are ignored when generating feature vectors for test instances. To avoid extremely large feature vectors, it is important to avoid colour explosion, where large numbers of colours are introduced over the iterations of the WL algorithm, as this can lead to intractably large feature vectors. Like Chen and Thiébaux (2024), when aggregating neighbours in the WL kernel, we use a set instead of the traditional multiset to avoid colour explosion.

Similarly, we make the same decision as Chen, Trevizan, and Thiébaux (2024) and Chen and Thiébaux (2024) to ignore static atoms in our graphs. However, unlike their work, we will use the colour of object nodes to represent some basic static information while keeping the number of colours low. Specifically, given a lifted planning task Π, we say a predicate is *static* if it does not appear in the effect of any action schema. We are particularly concerned with static predicates of arity 1. In both of our graphs, for each

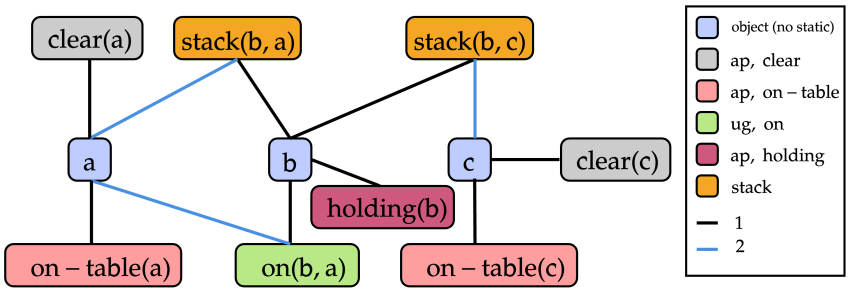


Figure 4.2: Example of an Action-Object-Atom Graph (AOAG) for a Blocksworld instance. Here, there are three blocks a, b, and c, with a and c on the table and b being held. The goal is to place b on a. Here the action set Λ includes all applicable actions instantiated from the stack action schema.

object o in the domain, we will give it a colour $\mathcal{P}_o$ that represents the static predicates of arity 1 that are true for o. This way, we are able to capture some basic static information in our graphs.

4.3.1 Action-Object-Atom Graph

Our first graph representation reflects the view that the actions in an action set are related through the objects that they act on. This leads to a straightforward extension of the Instance Learning Graph introduced in Section 2.3.3, where we additionally include nodes that represent the elements of the action set. We call this graph the *Action-Object-Atom Graph*. Its definition is given below, and an example is shown in Figure 4.2.

Definition 12 (Action-Object-Atom Graph)**.** *Given a state s and a set of actions Λ in a lifted planning task $\Pi = \langle\langle \mathcal{P}, \mathcal{A}\rangle, \langle O, s_0, G\rangle\rangle$, the* Action-Object-Atom Graph *(AOAG) depends on the action set Λ. If Λ contains only a single action a, then the AOAG is the ILG for the resulting state of applying a to s with static predicate colouring of object nodes described previously. Otherwise, if $\Lambda = \mathbb{A}_s$, then the AOAG is simply the ILG for the state s with static predicate colouring of object nodes discussed above.*

When neither of the two above special cases apply, the AOAG is the graph $\langle V, E, c, l\rangle$ where:

- *the vertices are those in the ILG with the addition of the nodes representing actions in Λ, i.e., $V = O \cup s \cup G \cup \Lambda$;*
- *the edges are those in the ILG with the addition of edges connecting actions in Λ*

to the objects used to instantiate them, i.e.,

$$E = \left(\bigcup_{p=P(o_1,\ldots,o_k)\in s\cup G} \{\langle p, o_1\rangle, \ldots, \langle p, o_k\rangle\} \right) \cup \left(\bigcup_{a=A(o_1,\ldots,o_k)\in\Lambda} \{\langle a, o_1\rangle, \ldots, \langle a, o_k\rangle\} \right)$$

- *the vertex colouring function c is similar to that of the ILG. Specifically, for all $o \in O$, $c(o) = \mathcal{P}_o$ where $\mathcal{P}_o$ is the set of static predicates of arity 1 that apply to o; for all $p = P(o_1, \ldots, o_k) \in s \cup G$, its colour is the same as in the ILG, i.e.,*

$$c(p) = \begin{cases} (\text{ap}, P), & \text{if } p \in s \setminus G \\ (\text{ag}, P), & \text{if } p \in s \cap G \\ (\text{ug}, P), & \text{if } p \in G \setminus s \end{cases}$$

  and for all $a = A(o_1, \ldots, o_k) \in \Lambda$, its colour is simply A.

- *The edge labelling function l is again similar to that of the ILG. Specifically, for all $\langle p, o_i\rangle \in E$ where $p \in s \cup G$ and $o_i \in O$, $l(\langle p, o_i\rangle) = i$, and for all $\langle a, o_i\rangle \in E$ where $a \in \Lambda$ and $o_i \in O$, $l(\langle a, o_i\rangle) = i$.*

The AOAG is a natural extension of the ILG of a shallow encoding of the actions in the action set. By connecting the actions to their instantiating objects, the graph indirectly establishes relationships on how they may change the state. By colouring the action nodes with colours representing their action schema, the resulting learned heuristic can potentially capture the structure of the action schema.

It is worth explaining the special cases when the action set Λ is just a single action or the set of all applicable actions $\mathbb{A}_s$. Here, the AOAG devolves into the ILG of either the current state or the resulting state of applying the single action. This highlights the fact that different graph representations may be used together, as long as their colours are compatible. Here, since the AOAG is an extension of the ILG, their colours can be used together. By using this property in these special cases, we are able to directly represent what the action set means at the state level, and thereby save the need to introduce additional nodes for the action set. For example, in the case where $\Lambda = \mathbb{A}_s$, we know that the action set is as large as it could be in the state s, and hence we can directly use the ILG of s.

4.3.2 Action Effect Graph

Our second graph representation is a deeper encoding of the actions in the action set than the AOAG. Specifically, we view that these actions each represent an option of how to change the state. Some effects of these actions may be shared by all of them, and are

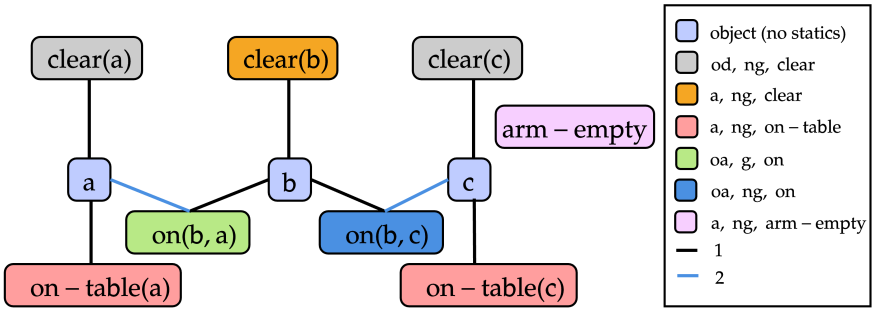


Figure 4.3: Example of an Action Effect Graph (AEG) for a Blocksworld instance. Here, there are three blocks a, b, and c, with a and c on the table and b being held. The goal is to place b on a. Here the action set Λ includes all applicable actions instantiated from the stack action schema.

hence guaranteed to happen. Other effects only exist in some of these actions, and are hence optional. This leads to our second graph representation, the *Action Effect Graph*. Its definition is given below, and an example is shown in Figure 4.3.

Definition 13 (Action Effect Graph)**.** *Given a state s and a set of actions Λ in a lifted planning task $\Pi = \langle\langle \mathcal{P}, \mathcal{A}\rangle, \langle O, s_0, G\rangle\rangle$, the* Action Effect Graph*(AEG) depends on the action set Λ. If Λ is the set of all applicable actions $\mathbb{A}_s$, then the AEG is the ILG for the state s with static predicate colouring of object nodes described previously.*

Otherwise, we consider the following sets of atoms:

- *the set of* necessary add effects $\mathrm{nec}_{\mathrm{add}}(\Lambda)$*, defined by* $\mathrm{nec}_{\mathrm{add}}(\Lambda) = \bigcap_{a\in\Lambda} \mathrm{add}(a)$*;*
- *the set of* necessary delete effects $\mathrm{nec}_{\mathrm{del}}(\Lambda)$*, defined by* $\mathrm{nec}_{\mathrm{del}}(\Lambda) = \bigcap_{a\in\Lambda} \mathrm{del}(a)$*;*
- *the set of* optional add effects $\mathrm{opt}_{\mathrm{add}}(\Lambda)$*, defined by* $\mathrm{opt}_{\mathrm{add}}(\Lambda) = \bigcup_{a\in\Lambda} \mathrm{add}(a) \setminus \mathrm{nec}_{\mathrm{add}}(\Lambda)$*;*
- *the set of* optional delete effects $\mathrm{opt}_{\mathrm{del}}(\Lambda)$*, defined by* $\mathrm{opt}_{\mathrm{del}}(\Lambda) = \bigcup_{a\in\Lambda} \mathrm{del}(a) \setminus \mathrm{nec}_{\mathrm{del}}(\Lambda)$.

To define the AEG, we will use the state s' given by applying all the necessary effects to s, i.e., $s' = (s \setminus \mathrm{nec}_{\mathrm{del}}(\Lambda)) \cup \mathrm{nec}_{\mathrm{add}}(\Lambda)$. For simplicity, we will also assume that $\mathrm{opt}_{\mathrm{del}}(\Lambda) \subseteq s'$ and that $\mathrm{opt}_{\mathrm{add}}(\Lambda) \cap s' = \emptyset$. These assumptions should hold for any well-defined planning task. The delete effects of any action in Λ should not delete atoms not in the state s, and the add effects of any action in Λ should not add atoms already in the state s. Moreover, the effects of any particular action should not both add and delete the same atom. It can be shown that our assumptions are a direct consequence of these conditions.

Then, the AEG is the graph $\langle V, E, c, l\rangle$ where:

- *The vertices are those in the ILG of s' with the addition of nodes representing the*

optional effects. Formally,

$$V = O \cup G \cup s' \cup \mathrm{opt}_{\mathrm{add}}(\Lambda).$$

Note that except for the objects, all other vertices are atoms.

- *The edges are given by connecting atoms to their argument objects, i.e.,*

$$E = \bigcup_{p=P(o_1,\ldots,o_k)\in G\cup s'\cup \mathrm{opt}_{\mathrm{add}}(\Lambda)} \{\langle p, o_1\rangle, \ldots, \langle p, o_k\rangle\}.$$

- *The vertex colouring function* $c : V \to 2^{\mathcal{P}} \cup (\{\mathrm{a}, \mathrm{u}, \mathrm{oa}, \mathrm{od}\} \times \{\mathrm{g}, \mathrm{ng}\} \times \mathcal{P})$ *is given by first mapping all object nodes to the set of static predicates of arity 1 that apply to them, and then for all atom nodes* $p = P(o_1, \ldots, o_k)$*, colouring them as* (α, β, P)*. The* α *component is determined by*

$$\alpha = \begin{cases} \mathrm{a}, & \text{if } p \in s' \setminus \mathrm{opt}_{\mathrm{del}}(\Lambda) \\ \mathrm{u}, & \text{if } p \in G \setminus (s' \cup \mathrm{opt}_{\mathrm{add}}(\Lambda)) \\ \mathrm{oa}, & \text{if } p \in \mathrm{opt}_{\mathrm{add}}(\Lambda) \\ \mathrm{od}, & \text{if } p \in \mathrm{opt}_{\mathrm{del}}(\Lambda) \end{cases}$$

*Here "*a*" stands for* ***a****chieved, "*u*" stands for* ***u****nachieved, "*oa*" stands for* ***o****ptional* ***a****dd, and "*od*" stands for* ***o****ptional* ***d****elete. An illustration of how* α *is determined is given in Figure 4.4.*

Compared to the α *component, the* β *component is simpler. Specifically,* β *is* g *if* $p \in G$ *and* ng *otherwise. Together, the* α *and* β *components determine which set of atoms* p *belongs to.*

- *The edge labelling function* l *is similar to that of the ILG. For all* $\langle p, o_i\rangle \in E$ *where* $p \in G \cup s' \cup \mathrm{opt}_{\mathrm{add}}(\Lambda)$ *and* $o_i \in O$*,* $l(\langle p, o_i\rangle) = i$*.*

By making the distinction between necessary and optional effects, the AEG is able to capture the structure of the action set in a more detailed manner than the AOAG. This is particularly useful in the case of partial space search, since the action sets induced by partial actions typically have a structure where some effects are shared by all actions, due to some parameters being fixed.

It is again worth discussing the special cases for the AEG. When the action set Λ is the set of all applicable actions $\mathbb{A}_s$, the AEG, like the AOAG, devolves into the ILG of the state s. Additionally, when the action set contains just a single action, all effects of the action are necessary. This way, the AEG naturally devolves into the ILG of the resulting state of applying the action, without the need to introduce a special case. Thus, the AEG has the same behaviour as the AOAG in these special cases.

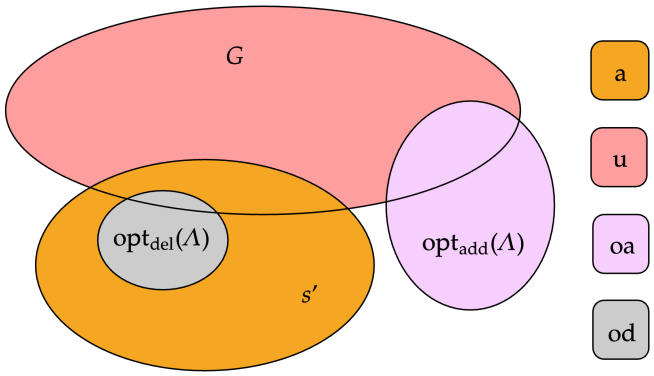


Figure 4.4: Venn diagram illustrating how the α variable in the AEG definition is determined. The set of atoms is constructed following the assumption that $\text{opt}_{\text{del}}(\Lambda) \subseteq s'$ and $\text{opt}_{\text{add}}(\Lambda) \cap s' = \emptyset$.

4.4 Training for Partial Space Search

Given the graph representations of state and action set pairs, we can now obtain feature vectors from state and action set pairs using the WL kernel. In this section, we discuss how to train action set heuristics using these feature vectors. Like in Section 2.3.4, we will discuss how to obtain action set heuristics both using regression and ranking as the heuristic target.

Before diving in, it is important to note that although the heuristics we obtain in the end are action set heuristics, the training process is done through partial space search. This means that the action set heuristics may be specialised to partial space search. In our use case where we do intend to use these heuristics for partial space search, this is an advantage. However, in other potential use cases of action set heuristics, this may be a drawback. We leave it to future work to investigate whether specialising action set heuristics for a particular use case has significant impacts on their performance in other use cases.

Similar to the training process for state space heuristics, we will assume that our training data consists of a set of lifted planning tasks $\Pi_1, \dots, \Pi_n$ on the same planning domain, with corresponding training plans $\pi_1, \dots, \pi_n$. Each such plan is a sequence of actions $\pi_i = a_1, \dots, a_{|\pi_i|}$ that solves the task Π_i by starting at the initial state s_0 and visiting the states $s_1, \dots, s_{|\pi_i|}$, where $s_{|\pi_i|}$ is a goal state. Such a plan represents a sequence of expansions in state space search. For our use case of partial space search, we decompose each action a_i into a sequence of partial actions of increasing specificity $\rho_{i,1}, \dots, \rho_{i,n_i}$. Here $\rho_{i,k}$ is the partial action of specificity $k-1$ that is a parent of a_i in the partial action tree, with $\rho_{i,1} = \text{None}$ and $\rho_{i,n_i} = a_i$. This way, we obtain a sequence of state and partial action sequence pairs

$$\langle s_0, (\rho_{1,1}, \dots, \rho_{1,n_1}) \rangle, \dots, \langle s_{|\pi_i|-1}, (\rho_{|\pi_i|,1}, \dots, \rho_{|\pi_i|,n_{|\pi_i|}}) \rangle.$$

Example 2. *Consider the Blocksworld plan* $\pi = \text{unstack}(a, b), \text{putdown}(a)$ *that starts in the state* s_0 *where* a *is on* b*, first picks up* a *yielding the state* s_1 *where* a *is held, and then places* a *down yielding the goal state* s_2 *where* a *is on the table. The resulting sequence of state and partial action sequence pairs is*

$$\langle s_0, (\text{None}, \text{unstack}(\_, \_), \text{unstack}(a, \_), \text{unstack}(a, b))\rangle,$$
$$\langle s_1, (\text{None}, \text{putdown}(\_), \text{putdown}(a))\rangle.$$

Note that in partial space search, we would actually directly expand the node $\langle s_0, \text{unstack}(a, b)\rangle$ *into* $\langle s_1, \text{putdown}(a)\rangle$ *because we skip over nodes that are unique successors. We do not do this when generating training data. This is because 1) we obtain more training data this way, 2) it simplifies the training process, and 3) similar nodes may be unique successors in some tasks (e.g., training tasks) but not in others (e.g., test tasks), ignoring them in our training data may lead to suboptimal heuristics.*

4.4.1 Regression

Using the state and partial action sequence pairs described above, we first discuss how to train a heuristic using regression. Given a state s and a partial action sequence $\rho_1, \ldots, \rho_n$, where the cost to reach the goal from s by following the training plan is c, we obtain a regression training dataset. Specifically, we uniformly distribute the cost γ of the ground action ρ_n amongst the partial actions, yielding the dataset,

$$\left\{(\phi(s, \rho_1), c), (\phi(s, \rho_2), c - \frac{1}{n}\gamma), \ldots, (\phi(s, \rho_n), c - \frac{n-1}{n}\gamma)\right\}.$$

Here, ϕ is the function that maps a pair of state and partial action first into their graph representations (either AOAG or AEG), then into a feature vector using the WL kernel. Each element of this dataset is a pair of a feature vector and their regression target, which is the cost to reach the goal from the state and partial action pair. Note that by construction ρ_1 is always None and ρ_n is always a ground action. Going from ρ_1 to ρ_n represents gradually refining the action set from all applicable actions to just ρ_n, and hence the cost of reaching the goal from ρ_1 to ρ_n is decreasing.

Our complete dataset $\mathcal{D}$ is then the union of such datasets for all state and partial action sequence pairs in the training plans. We then train a standard regression model using this dataset to minimise the mean squared error,

$$L(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \left(y - \mathcal{H}_\theta(x)\right)^2,$$

where $\mathcal{H}_\theta$ is the regression model (i.e., our action set heuristic) with parameters θ. In theory, $\mathcal{H}_\theta$ can be any regression model. In practice, we use the same Gaussian Process Regression model as in Chen, Trevizan, and Thiébaux (2024).

Example 3. *We continue from Example 2. We assume that the cost of each action is 1. Given the state and partial action sequence pairs from the example, our complete regression dataset $\mathcal{D}$ consists of the following elements*

$$\left(\phi(s_0, \text{None}), 2\right), \qquad \left(\phi(s_0, \text{unstack}(\_, \_)), \frac{7}{4}\right),$$
$$\left(\phi(s_0, \text{unstack}(a, \_)), \frac{3}{2}\right), \qquad \left(\phi(s_0, \text{unstack}(a, b)), \frac{5}{4}\right),$$
$$\left(\phi(s_1, \text{None}), 1\right), \qquad \left(\phi(s_1, \text{putdown}(\_)), \frac{2}{3}\right),$$
$$\left(\phi(s_1, \text{putdown}(a)), \frac{1}{3}\right).$$

4.4.2 Ranking

We have discussed the importance of ranking-based heuristics in Section 2.3.4. Here, we discuss how to train a ranking-based action set heuristic from the state and partial action sequence pairs. Our dataset will consist of tuples of the form $\langle \mathbf{x}, \mathbf{x}', \sigma, \delta \rangle$. Such a tuple represents the relation that the heuristic should rank the feature vector $\mathbf{x}$ lower than $\mathbf{x}'$ by a gap of at least δ, with the importance of this relation being σ. Here by gap, we mean that if a heuristic value of α is given to $\mathbf{x}'$, then the value given to $\mathbf{x}$ should be at most $\alpha - \delta$. In this section we will show how to generate such tuples for training action set heuristics.

Given a state s and a sequence of partial actions $\rho_1, \ldots, \rho_n$, we produce four types of tuples:

1. **Layer predecessors**, which rank later partial actions (higher specificity) better than earlier partial actions (lower specificity). Specifically, let the state before s be s' and the ground action applied in s' to reach s be a', then we produce the tuples

$$\{\langle \phi(s, \rho_1), \phi(s', a'), 1, \sigma_{\text{lp}} \rangle\}$$
$$\cup \bigcup_{i=2}^{n} \{\langle \phi(s, \rho_i), \phi(s, \rho_{i-1}), 1, \sigma_{\text{lp}} \rangle\},$$

   where ϕ is the same function as in the regression case, and σ_{lp} is the importance value for all layer predecessor tuples.

2. **Layer siblings**, which rank partial actions in the sequence better than other partial actions with the same specificity but not in the sequence. Specifically, for each $i = 1, \ldots, n$ and for each ρ', where ρ' is an applicable partial action in s of specificity $\text{spec}(\rho_i)$ that is not in the sequence, we produce the tuple,

$$\langle \phi(s, \rho_i), \phi(s, \rho'), 0, \sigma_{\text{ls}} \rangle,$$

where σ_{ls} is the importance value for all layer sibling tuples. Here the gap is 0 as we only know that the partial actions are no worse than their siblings, but not by how much.

3. **State predecessors**, which rank the state s better than the previous state s', and rank the partial actions in the sequence better than the state s (represented by the feature $\phi(s, \text{None})$). Specifically, we produce the tuples

$$\{\langle \phi(s, \text{None}), \phi(s', \text{None}), 1, \sigma_{\text{sp}} \rangle\}$$
$$\cup \bigcup_{i=2}^{n} \{\langle \phi(s, \rho_i), \phi(s, \text{None}), 1, \sigma_{\text{sp}} \rangle\},$$

where σ_{sp} is the importance value for all state predecessor tuples. Note that since $\rho_1 = \text{None}$, the state predecessor tuples have a lot in common with the layer predecessor tuples. They act similarly to the skip connections commonly seen in neural network architectures.

4. **State siblings**, which rank the fully instantiated partial action ρ_n better than other ground actions that are applicable in s. Specifically, let $\mathbb{A}_s$ be the set of all applicable actions in s, then for each $a \in \mathbb{A}_s$ that is not ρ_n, we produce the tuple

$$\langle \phi(s, \rho_n), \phi(s, a), 0, \sigma_{\text{ss}} \rangle,$$

where σ_{ss} is the importance value for all state sibling tuples. Here the gap is 0 as we only know that ρ_n is no worse than the other actions, but not by how much.

These four types of tuples from all state and partial action sequence pairs in the training plans form our complete dataset $\mathcal{D}$. Note that by including state siblings and predecessors, our dataset is effectively a superset of the dataset proposed in Hao et al. (2024) for training state space heuristics. There, they generated similar tuples for training state space heuristics. Given that they train state space heuristics, they only used tuples between each state and its predecessor state (i.e., our state predecessors) and between sibling states (i.e., our state siblings).

Given our complete dataset $\mathcal{D} = \{\langle \mathbf{x}_i, \mathbf{x}'_i, \sigma_i, \delta_i \rangle\}$, we learn a linear model by solving the following linear program,

$$\begin{aligned} \min_{\mathbf{w},\mathbf{z}} \quad & C \sum_i \sigma_i \mathbf{z}_i + \|\mathbf{w}\|_1 & \\ \text{s.t.} \quad & \mathbf{z}_i \geq 0 & \forall i \\ & \mathbf{w}^T(\mathbf{x}_i - \mathbf{x'}_i) \geq \delta_i - \mathbf{z}_i & \forall i \end{aligned}$$

Here, the vector $\mathbf{w}$ is the weight vector of the linear model, C is the regularisation parameter, and $\mathbf{z}$ is a vector of slack variables for each constraint generated from the ranking tuples. Note that this linear program is a variant of the standard RankSVM model (Joachims, 2002) with L_1 regularisation. Note that $\|\mathbf{w}\|_1$ is not itself a linear

function. However, it can be trivially encoded in a linear program through the use of a common trick. Specifically, we introduce a new variables $\mathbf{w}^+$ and $\mathbf{w}^-$, and add the constraint $\mathbf{w}^+, \mathbf{w}^- \geq 0$. This way, we use $\mathbf{w}^+ - \mathbf{w}^-$ as $\mathbf{w}$, and encode its L1-norm as $\mathbf{w}^+ + \mathbf{w}^-$.

Note also that the importance hyperparameters σ_{lp}, σ_{ls}, σ_{sp}, and σ_{ss} are important for training heuristics that generalise well. Typically, the resulting dataset is imbalanced, which may not yield good heuristics. We use these hyperparameters to manage the contribution of each type of tuple to the training process, and to ensure that the resulting heuristic generalises well.

Example 4. *We again continue from Example 2. Given the state and partial action sequence pairs from example, we generate the layer predecessors tuples,*

$$\begin{aligned}
&\langle \phi(s_0, \text{unstack}(_, _)), \phi(s_0, \text{None}), 1, \sigma_{\text{lp}} \rangle, \\
&\langle \phi(s_0, \text{unstack}(a, _)), \phi(s_0, \text{unstack}(_, _)), 1, \sigma_{\text{lp}} \rangle, \\
&\langle \phi(s_0, \text{unstack}(a, b)), \phi(s_0, \text{unstack}(a, _)), 1, \sigma_{\text{lp}} \rangle, \\
&\langle \phi(s_1, \text{None}), \phi(s_0, \text{unstack}(a, b)), 1, \sigma_{\text{lp}} \rangle, \\
&\langle \phi(s_1, \text{putdown}(_)), \phi(s_1, \text{None}), 1, \sigma_{\text{lp}} \rangle, \\
&\langle \phi(s_1, \text{putdown}(a)), \phi(s_1, \text{putdown}(_)), 1, \sigma_{\text{lp}} \rangle;
\end{aligned}$$

and the layer sibling tuples,

$$\begin{aligned}
&\langle \phi(s_1, \text{putdown}(_)), \phi(s_0, \text{stack}(_, _)), 1, \sigma_{\text{lp}} \rangle, \\
&\langle \phi(s_1, \text{putdown}(a)), \phi(s_0, \text{stack}(a, _)), 1, \sigma_{\text{lp}} \rangle;
\end{aligned}$$

and the state predecessors tuples,

$$\begin{aligned}
&\langle \phi(s_0, \text{unstack}(_, _)), \phi(s_0, \text{None}), 1, \sigma_{\text{sp}} \rangle, \\
&\langle \phi(s_0, \text{unstack}(a, _)), \phi(s_0, \text{None}), 1, \sigma_{\text{sp}} \rangle, \\
&\langle \phi(s_0, \text{unstack}(a, b)), \phi(s_0, \text{None}), 1, \sigma_{\text{sp}} \rangle, \\
&\langle \phi(s_1, \text{None}), \phi(s_0, \text{None}), 1, \sigma_{\text{sp}} \rangle, \\
&\langle \phi(s_1, \text{putdown}(_)), \phi(s_1, \text{None}), 1, \sigma_{\text{sp}} \rangle, \\
&\langle \phi(s_1, \text{putdown}(a)), \phi(s_1, \text{None}), 1, \sigma_{\text{sp}} \rangle;
\end{aligned}$$

and the single state siblings tuple,

$$\langle \phi(s_1, \text{putdown}(a)), \phi(s_1, \text{stack}(a, b)), 1, \sigma_{\text{ss}} \rangle.$$

Chapter 5

Evaluation

In Chapter 3 we had introduced partial space search, a novel search space with reduced branching factor that is particularly well-suited for learning for planning. In Chapter 4, we then discussed how to obtain both learned and automatically translated action set heuristics for guiding partial space search. In this chapter, we first discuss in Section 5.1 the new planning system, LazyLifted, where we implemented our contributions. We discuss the efficiency, maintainability, and testing of this planning system, and how it supports the PDDL language. We then discuss in Section 5.2 the benchmark domains that we used to evaluate LazyLifted. Here, we use both existing competition domains from the International Planning Competition, and new domains designed to test planning system under high branching factor conditions. In Section 5.3, we discuss our experimental methodology, including hardware and software used, training and testing instances, baselines, and the metrics used to evaluate the planning systems. Finally, in Section 5.4, we present the results of our experiments and analyse them in detail to fully understand the empirical value of our contributions.

5.1 Implementation of the LazyLifted Planning System

Typically, planning research is implemented by extending existing mature and well-optimised planning systems. For classical planning, this is likely the Fast Downward planning system (Helmert, 2006), which is widely used and actively maintained. As examples, the planners LAMA (Richter and Westphal, 2010) and Scorpion (Seipp, 2023), and the learning for planning system GOOSE (Chen, Trevizan, and Thiébaux, 2024) are all based on Fast Downward.

A major limitation of Fast Downward is that it always grounds the lifted planning task. As discussed in Section 2.2.2, some planning tasks can be hard to ground and require alternative approaches. For this, the lifted planning system Powerlifted has

been developed (Corrêa et al., 2020). From an architectural perspective, Powerlifted is largely based on Fast Downward. This gives it a solid performant foundation, but also means it inherits various technical debts from Fast Downward in addition to its own limitations. Most prominently, Powerlifted does not support negative preconditions, a common feature in many planning domains. Moreover, Powerlifted inherits an old and arcane translator for parsing PDDL files into an internal representation.

In this thesis, our goal is to ultimately develop a scalable learning for planning system using partial space search and action set heuristics. The scalability requirement meant that we would like to use a lifted planning system that supports large and hard to ground planning tasks. However, the implementation of partial space search also meant that we would need to develop much of the planning system from scratch. In other words, the benefits to basing our implementation on an existing planning system is limited.

Given these considerations, we developed a new planning system called LazyLifted (LL). As its name suggests, LazyLifted is a lifted planning system whose architecture is based on Powerlifted. However, LazyLifted is a Rust rewrite of Powerlifted that is more modern, maintainable, and scalable. In other words, LazyLifted is production-ready, and we believe new planning research work should seriously consider LazyLifted as where they implement their work. Throughout this chapter, we do not aim to provide a comprehensive overview of LazyLifted, but instead to highlight its key features. In Section 5.1.1, we discuss the efficiency and scalability of LazyLifted. In Section 5.1.2, we discuss the work we have done to make LazyLifted more maintainable. In Section 5.1.3, we discuss how the critical components of LazyLifted are tested to ensure correctness. In Section 5.1.4, we discuss the PDDL subsets supported by LazyLifted and how this support is achieved. Lastly, in Section 5.1.5, we discuss a dependency of LazyLifted that we developed to train action set heuristics efficiently.

5.1.1 Efficiency and Scalability

The forefront goal of any planner is to be efficient and scalable. Consequently, we implemented LazyLifted with a focus on performance. The main way we achieved this is by basing our implementation on Powerlifted, which is itself based on Fast Downward. This provides a solid foundation for LazyLifted and inherit the benefits of many of the performance related work done in these systems. We additionally implement several performance improvements. These improvements are based on profiling LazyLifted on real planning tasks and identify unnecessary computation. The flamegraph[1] tool provided significant help in identifying rooms for improvement and was very easy to use. Here we discuss the important performance improvements we made to LazyLifted over the existing planning systems, and evaluate their impact empirically.

Cached unpacking. As a memory optimisation, both Powerlifted and Fast Downward uses two memory representations of planning states. The first is an unpacked represen-

[1] Available at https://github.com/flamegraph-rs/flamegraph.

tation that is easy to access and manipulate, but expensive on memory. The second is a packed representation that is much more memory efficient. This requires a packing and unpacking step when switching between the two representations. In LazyLifted, we inherit these two memory representations. However, we identified that in certain tasks, the unpacking step can occur so frequently that it consumes around 30% of the total runtime. To alleviate this, we add a simple Least Recently Used (LRU) cache to store a mapping from the packed representation to the unpacked representation for up to 1000 states. Whenever we unpack a state, we first check if it is in the cache and if so, we use the unpacked representation from the cache without performing the unpacking step. Profiling on a few tasks shows that this simple change almost removes the unpacking overhead entirely. The implementation of this is made easy through the `lru` library[2].

Small vectors. In various parts of the codebase, an important data structure is a small vector containing a few small integers. An example occurrence of this is the representation of ground atoms, which involves a vector of integers representing the objects in the atom. In Rust, the default memory representation of a vector is a pointer to a heap-allocated array, which can mean many tiny allocations for small vectors. To alleviate this, we use the `SmallVec` library[3] to store the first few elements of the vector alongside the pointer directly. Thanks to the small size of these vectors in most cases, this usually removes the need for a heap allocation entirely.

Internment. Internment is a technique to reduce memory usage by storing a single copy of each unique value. This is most useful when the values are immutable, and it is expected that many copies of the same value will be created. Such is exactly the case with many of the small vectors we had just discussed. As such, we used the `internment` library[4] to store only the unique small vectors and reuse them when needed. This reduced the memory usage of LazyLifted significantly, and also improves performance by making it easy to compare small vectors, since it is now just a pointer comparison.

Altogether, these performance improvements make LazyLifted more efficient both in speed and memory usage. We evaluate their impacts by running LazyLifted and Powerlifted with the exact same algorithm and heuristics and comparing their respective run times and memory usage. The complete detail of our experimental methodology and benchmark domains are discussed in the later sections, but we present the results now for coherence.

Figure 5.1 shows the results of LazyLifted and Powerlifted on our complete benchmark set. As a guide for understanding the results, points on the top-left favour LazyLifted and points on the bottom-right favour Powerlifted. Note that Powerlifted fails to solve many tasks due to its lack of support for negative preconditions.

[2]Available at https://github.com/jeromefroe/lru-rs.
[3]Available at https://github.com/servo/rust-smallvec.
[4]Available at https://github.com/droundy/internment.

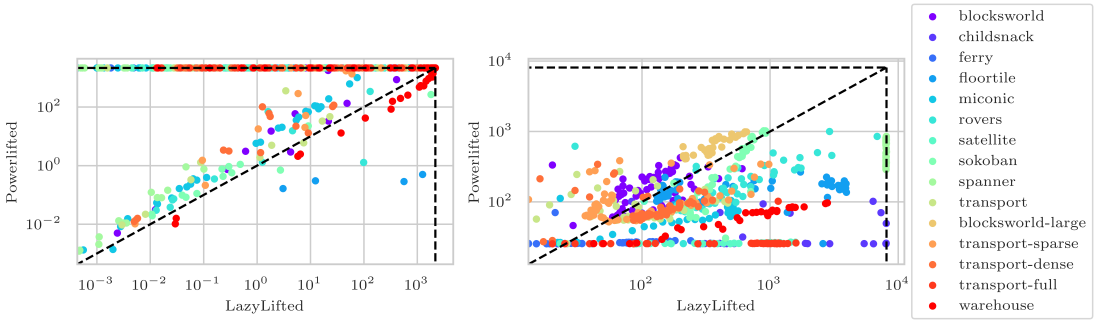


Figure 5.1: Comparison of LazyLifted and Powerlifted on our complete benchmark set with both using Greedy Best First Search (GBFS) with the FF heuristic on state space search. The run time results in seconds are shown on the left, and the memory usage results in megabytes (MB) are shown on the right. If a planner fails to solve a task, its time is set to the maximum limit of 1800 seconds. LazyLifted only reports memory usage if it runs for at least 10 seconds, we do not show memory usage when it does not report it.

It is apparent from the runtime results that LazyLifted is generally noticeably more performant than Powerlifted with a few exceptions on the floortile domain. However, LazyLifted is not always more memory efficient than Powerlifted. We believe this means additional room for memory optimisation in LazyLifted, in particular by investigating the memory optimisations done in Powerlifted and implementing them in LazyLifted. We leave this as future work. However, this does not mean the internment optimisation is not useful. Anecdotally, we had observed that the internment optimisation was able to reduce the memory usage of LazyLifted by up to 80%. This in turn allowed it to solve many large problems where it would previously have run out of memory.

5.1.2 Maintainability

We aim for LazyLifted to be an easy base planner for future research work to be implemented in. As such, this means that the code should be easy to understand, extend, and modify. Generally, this means following best practices in software engineering. Specific to planning, maintainability is made easier by following the general architecture of the existing Powerlifted and Fast Downward planners, which many other researchers are already familiar with. However, we also make deviations to their architecture when beneficial, this in particular is made possible by using type system features in Rust.

Unlike Powerlifted, LazyLifted intentionally avoids using implementation inheritance. There has been argument in support of this design decision for decades, with a notable example being the "Gang of Four" design patterns book (Gamma et al., 1994), which argued for "program to an interface, not an implementation" and "composition over inheritance". Rust in fact enforces this by not having implementation inheritance at

all. Instead, Rust provides traits (similar to interfaces in Java) and enums (similar to sum types in functional programming languages) to achieve the same goals. In our experience, this has made the codebase much easier to understand and extend.

Another key aspect of maintainability is the use of modern tooling. Rust has a rich ecosystem of tools that help with code quality. For example, the `clippy` tool[5] provides lints that help catch common mistakes and improve code quality. The `rustfmt` tool[6] automatically formats the code to a consistent style, which is particularly useful in a large codebase with many contributors. The `cargo` tool[7] automatically manages dependencies and builds the codebase, which makes it easy to get started with LazyLifted. This is especially in contrast to many C++ based projects where dependency management and building in uncommon platforms can be very challenging. LazyLifted uses all these tools to ensure a high quality codebase that is easy to maintain. Compared to their C++ counterparts, these tools generally offer a better user experience and are more reliable.

5.1.3 Testing

A key problem with many planning systems is that there is limited testing in their codebase. This is not only an issue for correctness, but also an obstacle for future research work. Changes required for new research work cannot be confidently made if there are no unit tests to ensure that the changes do not break existing functionality. In LazyLifted, we aim to have a comprehensive test in critical components of the codebase. This includes the PDDL parser, and the core successor generation algorithm. Correctness of these components ensures that the plans generated are correct. The test coverage of the successor generator components are shown in Figure 5.2, which shows that we have achieved a high level of test coverage in these components. The tests for the Parser are written in the form of doctests, which serve as both documentation and tests, but unfortunately are not detected by the coverage tool.

5.1.4 PDDL Support

The PDDL language is the de facto standard for describing planning tasks. It contains a rich set of features that can be used to describe a wide variety of planning tasks. Most planners tend to support a subset of PDDL, which they focus on. An unfortunate choice in Powerlifted is its lack of support for negative preconditions in action schemas, which are a relatively common feature in classical planning domains.

LazyLifted supports all PDDL features that Powerlifted supports, and additionally supports negative preconditions. This is achieved by compiling the negative preconditions away at run time through introducing additional predicates that represent the negation of existing predicates.

[5] Available at https://github.com/rust-lang/rust-clippy.
[6] Available at https://github.com/rust-lang/rustfmt.
[7] Available at https://github.com/rust-lang/cargo.

search/successor_generators/full_reducer.rs	100.00% (10/10)	93.10% (135/145)	91.01% (81/89)
search/successor_generators/hypergraph.rs	100.00% (1/1)	95.45% (42/44)	90.00% (18/20)
search/successor_generators/join_algorithm.rs	83.33% (15/18)	84.13% (159/189)	80.61% (79/98)
search/successor_generators/join_algorithm_tests.rs	100.00% (13/13)	100.00% (300/300)	100.00% (81/81)
search/successor_generators/join_successor_generator.rs	83.33% (15/18)	76.07% (178/234)	71.43% (75/105)
search/successor_generators/successor_generator.rs	50.00% (1/2)	83.33% (5/6)	57.14% (4/7)
search/task.rs	85.71% (42/49)	92.85% (558/601)	90.19% (193/214)
search/validate.rs	100.00% (5/5)	100.00% (82/82)	100.00% (17/17)
search/database/hash_join.rs	100.00% (13/13)	100.00% (108/108)	100.00% (72/72)
search/database/hash_semi_join.rs	100.00% (6/6)	100.00% (65/65)	100.00% (46/46)
search/database/join.rs	100.00% (10/10)	100.00% (98/98)	100.00% (71/71)
search/database/project.rs	100.00% (3/3)	100.00% (41/41)	100.00% (29/29)
search/database/semi_join.rs	100.00% (4/4)	100.00% (55/55)	100.00% (51/51)
search/database/table.rs	100.00% (2/2)	100.00% (9/9)	100.00% (2/2)
search/database/utils.rs	100.00% (2/2)	100.00% (20/20)	100.00% (19/19)

Figure 5.2: Test coverage of critical components of LazyLifted. The three numerical columns are function, line, and region coverage respectively.

5.1.5 The Rank2Plan Dependency

In order to train the action set heuristics we had introduced in Chapter 4, we developed a Python library that is used as a dependency for LazyLifted. This library is called Rank2Plan, and is used to more generally train L1-regularised RankSVMs, which are exactly the models we use for training ranking heuristics.

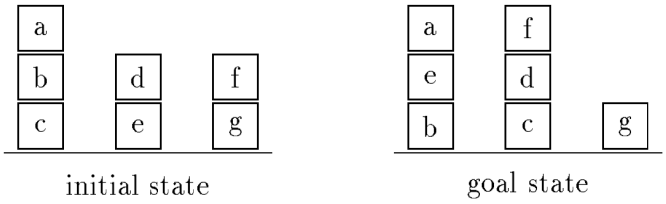
Rank2Plan does so by using constraint and column generation techniques introduced in Dedieu, Mazumder, and Wang (2022). We direct interested readers to this paper for a more detailed explanation of the techniques used. The library is a rewrite of their code to be easier to use and more maintainable. It also adds specialised support for RankSVMs, where their work focused on the more general Support Vector Machine (SVM) case.

In addition to this, we also implemented support for using Bayesian optimisation (Snoek, Larochelle, and Adams, 2012) to tune the hyperparameters of the RankSVM. Bayesian optimisation is a powerful technique for optimising expensive black-box functions, which is exactly the case for tuning the hyperparameters of the RankSVM. We implemented this efficiently reusing work done in previous hyperparameter trials in future trials.

We do not have thorough experimental results for Rank2Plan. Anecdotally, prior to using Rank2Plan, our training process could take up to 20 hours with a memory usage close to 28 GB. With Rank2Plan, the training process takes around up to 4 hours with a memory usage less than 20 GB. This is a significant improvement, particularly in time, and allows us to perform our experiments more efficiently.

5.2 Benchmark Domains

To evaluate our contributions, we use two sets of benchmark domains. The first set (IPC23-LT) is the benchmarks used in the International Planning Competition (IPC) 2023 Learning Track (Taitler et al., 2024). The second set (HBF) is a set of domains we have designed to test planning systems under high branching factor conditions. We



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 5.3: An example of a Blocksworld instance, taken from Slaney and Thiébaux (2001). This is the exact same figure as Figure 2.1.

describe these two set of benchmarks in Sections 5.2.1 and 5.2.2 respectively. After describing them, we then discuss the characteristics, such as branching factor, of these domains in Section 5.2.3.

5.2.1 International Planning Competition Domains

The International Planning Competition (IPC) is a series of competitions seeking to evaluate the state-of-the-art in planning. The latest competition, IPC 2023, featured a learning track specifically for learning for planning systems. This track featured a set of ten classical planning benchmark domains. For each domain, there are a set of training and testing instances. The training instances are designed to be used to train the learning for planning system, and the testing instances are used to evaluate how well these systems are able to generalise. Each test set is additional split into three groups: easy, medium, and hard. The easy group is roughly the same difficulty as the training instances, with the medium and hard groups being increasingly more difficult. In this section we describe the ten domains used in the IPC 2023 Learning Track. We leave a discussion of their sizes and difficulties to Section 5.2.3.

Blocksworld Blocksworld is a well-known planning domain consisting of a set of blocks stacked to form towers. Blocks can be picked up and then placed either on other blocks or on the table. The goal is to form a set of towers with a specific configuration of blocks. Amongst the variants of Blocksworld, the IPC 2023 Learning Track uses the 4-operation variant of Blocksworld, where feature four action schemas:

- `pick-up` a block from the table, when there is no block on top of it and no block is currently held;
- `put-down` a block on the table, when the block is currently held;
- `stack` a block on top of another block, when the block is currently held and there is no block on top of the block to be stacked on;

- `unstack` a block from the top of another block, when no block is currently held and there is no block on top of the block to be unstacked.

A simple strategy to solve any Blocksworld instance is to simply first place all blocks to the table, then stack them in the correct order. This strategy provides an upper bound on the difficulty of the domain and shows that every task can be solved. However, finding better plans can be challenging, in particular as solving Blocksworld optimally is NP-hard (Gupta and Nau, 1992). A more comprehensive study of the Blocksworld domain, including efficient algorithms for solving it satisficingly and optimally, can be found in Slaney and Thiébaux (2001).

Childsnack The Childsnack domain concerns planning to make and serve sandwiches to a group of children, some of whom may be allergic to gluten. It features three groups of action schemas:

- Sandwich making, with one action schema for making a sandwich, and another one for making a gluten-free sandwich;
- Sandwich serving, with one action schema for serving a sandwich, and another one for serving a gluten-free sandwich;
- Moving sandwiches, with one action schema to placing a sandwich on a tray, and another for moving the tray.

The goal of the domain is to serve sandwiches to the children. The action schemas are defined in a way that a gluten-free sandwich cannot contain any gluten contents, and a sandwich that is not gluten-free cannot be served to a child who is allergic to gluten.

This domain has two main difficulties. The first is the high degree of symmetry in this domain, since many sandwich ingredients, children, and sandwiches are equivalent from a planning perspective. These equivalences create a vast number of states and actions that are effectively the same, but planners that are not symmetry aware must reason through them separately. The second difficulty is the long-horizon nature of sandwich making and serving — a planner could easily decide to many non-gluten-free sandwiches, leaving insufficient ingredients to make gluten-free sandwiches for the allergic children. Such a mistake would only reveal itself after many actions have been taken, making it hard to correct.

Ferry Ferry is a domain where a ferry must transport cars between a number of locations. The cars each start at a particular location, and must be transported each to a specific destination. The ferry can only carry a single car at a time, and can move from any location to any other location in a single action. An example of this domain is shown in Figure 5.4.

Ferry is a reasonably easy domain, thanks to the fact that the ferry can move between any two locations in a single action. This removes the path finding element of the domain — the ferry can always just load any car that is not at its destination and move

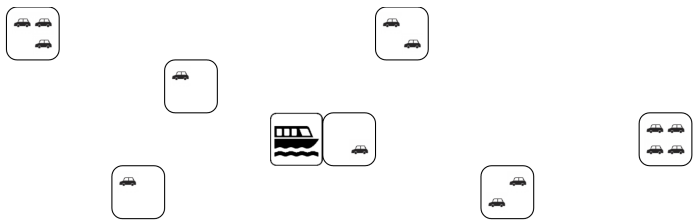


Figure 5.4: An example of a Ferry instance.

it to its destination. A potential challenge to planners from this domain is that large problem instances can result a very large space of reachable states and make this task hard-to-ground or memory consuming.

Floortile Floortile is a domain where a number of robot needs to move on a grid of tiles and paint these tiles to either black or white. Each robot can only paint the tile directly above or below it and can only move to an adjacent tile. The goal is to paint the tiles in a specific pattern.

Miconic Miconic is a domain where an elevator must determine how to best move between a number of floors to pick up and drop off passengers. The elevator can move between any two floors in a single action, pickup passengers not at their destination, and drop off passengers at their destination. The goal is to move all passengers to their destination floors. It is very similar in nature to the ferry domain, with the key difference being that the elevator can hold any number passengers at a time, unlike the ferry which can only hold one car at a time.

Rovers The Rovers domain is a simplified version of problems that confronted NASA for their Mars exploration missions. It involves several rovers (e.g., the Opportunity rover seen in Figure 5.5) that must traverse the Mars surface. Each rover is equipped with a set of equipment, which it can use to gather data and transmit it back to a lander. Each rover can only traverse of certain terrain types, meaning different parts of the planet are only accessible to certain rovers. Data transmission is also constrained by the visibility of the lander from the waypoints.

Satellite Like Rovers, Satellite is also a domain simplified from the satellite observation scheduling problem by NASA. In this domain, a number of satellites are each equipped with a set of instruments. Instruments should be calibrated to observe specific targets. The satellites can only observe targets when they are pointing in the right direction.

Sokoban The Sokoban domain is based on a classic video game where a player has to push boxes around a warehouse to their target locations. The warehouse is a square grid

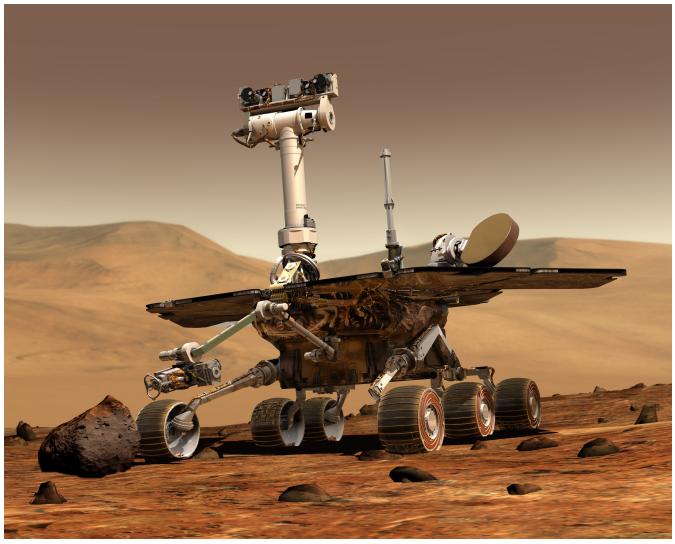


Figure 5.5: The Opportunity rover, one of the rovers used in the Mars exploration missions.

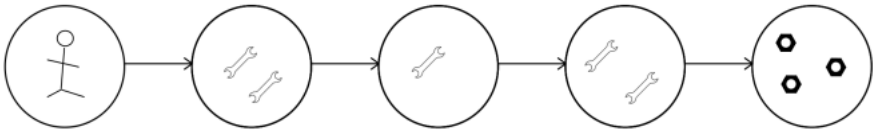


Figure 5.6: An example of a Spanner instance with 5 spanners, 3 nuts, and 3 locations, from Chen (2023)

with walls that box or player cannot pass through. In each step, the player can either move to an adjacent cell or push a box in the direction of the move. Since boxes cannot be pulled, deadends exist in this domain, where a box is stuck in a corner and cannot be moved to its target location. Research has shown that this problem is PSPACE-complete (Culberson, 1997), even if there are no walls in the warehouse (Hearn and Demaine, 2005).

Spanner The Spanner domain involves an agent (Bob) who must move within a one-way corridor, pickup spanners, and use these spanners to tighten nuts. The nuts are located on a gate at the end of a corridor. Each spanner can only be used to tighten one nut. The goal is to tighten all nuts on the gate. An example is shown in Figure 5.6.

This domain is particularly challenging for delete relaxation heuristics, as they cannot model the fact that a spanner can only be used once.

Transport The Transport domain involves a number of trucks that must navigate a graph of locations to transport packages to their destinations. Each truck has a capacity limit of how many packages it can carry. The main challenges of this domain are managing the capacity limits and the path finding problem of navigating the graph.

5.2.2 High Branching Factor Domains

The domains used in the IPC 2023 Learning Track provide a good variety of tasks to test learning for planning systems. However, in this thesis we are particularly interested in examining the performance impacts of partial space search in planning tasks with high branching factors. To this end, we have designed a set of new domains with high branching factors.

Blocksworld Large Generally, planning tasks do not include objects that are irrelevant to the goal. However, a desirable property for planners is the ability to effectively work with information irrelevant to the goal. Correspondingly, there has been a growing trend in planning research to learn models that can automatically produce planning task descriptions given, for example, visual traces of plans (Xi, Gould, and Thiébaux, 2024). These systems may not always learn to filter out irrelevant information.

As such, we have designed a varied set of tasks for the Blocksworld domain where there are many blocks that are irrelevant to the goal. These large blocks induce a large branching factor in general. To effectively complete these tasks, a planner must be able to either ignore the irrelevant blocks or deal with the branching factor induced by them.

Transport Sparse, Dense, and Fully Connected Another interesting thing to investigate is the impact of the graph structure on the Transport on branching factor and consequently the performance of planning systems. By density of a graph, we mean the ratio of the number of edges to the number of possible edges. A sparse graph has a low density, a dense graph has a high density, and a fully connected graph is where all possible edges are present. As the density of a graph increases, the branching factor of the Transport planning task increases as the trucks can move to more locations in one step. In the Transport domain, since there are also many trucks, the branching factor is roughly the number of trucks times the average degree of the locations in the graph.

Given this, we have designed three variants of the Transport benchmark set with sparse, dense, and fully connected graphs. Let V be the number of locations in the graph, then the maximum possible number of edges is $V(V-1)$. We ensure the graph is always connected by having at least $V-1$ edges. For the sparse graph, the number of edges is between $V-1$ and $1.5(V-1)$. For the dense graph, the number of edges is between 0.5 and 0.8 of the maximum possible. Lastly, for the fully connected graph, the number of edges is the maximum possible.

Warehouse Warehouse is the only new domain that we introduce that does not exist before to our knowledge. In this domain, there are blocks stacked together to form towers, similar to Blocksworld. Unlike Blocksworld, where an arm is used to move the blocks, in Warehouse blocks on top can be moved between towers in a single action. Moreover, there is a limit on the maximum number of towers. The goal of Warehouse is remove exactly a subset of blocks, i.e., taking them out of the warehouse.

The Warehouse domain is designed to have a high branching factor. In particular, since blocks can be moved between towers in a single action, the branching factor is quadratic to the number of towers. This allows us to investigate how well planners are able to effectively navigate this form of high branching factor.

5.2.3 Domain Characteristics

We use this section to analyse the characteristics of the domains we have in our benchmark sets. For each domain, there are 90 to 100 training instances and 90 test instances. The test set is additionally divided into three splits: easy, medium, and hard. In general, the training instances and easy test instances are roughly on the same scale. The medium and hard test instances then increase rapidly in size and hence difficulty. To describe the test set sizes, we use two metrics: the number of key objects and the branching factor. The former provides an intuitive indication of the size of the task, while the latter provides an indication of the computation demand of the tasks. Branching factor is particularly interesting due to our contribution of partial space search. We would like to investigate how partial space search handles tasks with varying branching factors.

Although branching factor is an intuitive concept, its measurement can be tricky. We measure branching factor as the number of state evaluations per state expansion in state space search. This is a good approximation of how much work planners need to do on average when expanding a single state. For any given task, the branching factor is influenced by the part of the search space explored and therefore differs for different heuristics and search algorithms. We use the FF heuristic and Greedy Best First Search (GBFS). Given that the FF heuristic is relatively good, this provides a relatively good approximation of the branching factor of the tasks that would be encountered in practice.

The results for the number of key objects and branching factor for the domains in our benchmark sets are shown in Table 5.1. It is worth noting that even some IPC domains, such as Childsnack and Satellite, have high branching factors. On the other hand, the high branching factor domains we have added consistently have high branching factors.

It is particularly worth noting that the transport domains in the HBF set have branching factors that increase with respect to graph density, as we had expected. It is also important to point out that the high branching factor in Childsnack is mostly due to the high degree of symmetry in the domain. This means that unlike most high branching factor cases, in Childsnack the high number of options faced by planners lead to small number of unique states.

Table 5.1: Description of test sets for each benchmark domain and size and branching factor. We only show the number of key objects when describing size. See text for explanation on branching factor.

Set	Domain	Split	Key Object Sizes	Branching Factor
IPC 2023 LT	blocksworld	test (easy)	5 to 30 blocks	1.71 to 6.29
		test (medium)	35 to 150 blocks	4.57 to 21.73
		test (hard)	160 to 500 blocks	19.71 to 60.96
	childsnack	test (easy)	4 to 10 children, 4 to 15 sandwiches	1.76 to 18.11
		test (medium)	15 to 40 children, 15 to 60 sandwiches	1680.71 to 26287.00
		test (hard)	50 to 300 children, 50 to 450 sandwiches	Expansion failure[a]
	ferry	test (easy)	2 to 20 cars, 5 to 15 locations	1.83 to 5.31
		test (medium)	10 to 100 cars, 20 to 50 locations	5.38 to 9.14
		test (hard)	200 to 1000 cars, 100 to 500 locations	20.50 to 223.50
	floortile	test (easy)	3 by 3 to 4 by 6 grid, 1 to 3 robots	1.89 to 3.47
		test (medium)	10 by 10 to 13 by 19 grid, 4 to 13 robots	5.61 to 73.17
		test (hard)	25 by 25 to 28 by 34 grid, 15 to 28 robots	82.58 to 212.42
	miconic	test (easy)	1 to 10 passengers, 4 to 20 floors	1.60 to 11.39
		test (medium)	20 to 80 passengers, 30 to 60 floors	17.43 to 38.74
		test (hard)	50 to 500 passengers, 80 to 200 floors	45.36 to 146.48
	rovers	test (easy)	1 to 4 rovers, 4 to 10 waypoints	3.77 to 30.15
		test (medium)	5 to 10 rovers, 15 to 90 waypoints	19.13 to 548.21
		test (hard)	15 to 30 rovers, 100 to 200 waypoints	338.36 to 726.00
	satellite	test (easy)	3 to 10 satellites	10.00 to 103.75
		test (medium)	15 to 40 satellites	199.45 to 1182.23
		test (hard)	50 to 100 satellites	1890.33 to 3532.00
	sokoban	test (easy)	8 by 8 to 13 by 13 grid, 1 to 4 boxes	1.25 to 2.83
		test (medium)	20 by 20 to 50 by 50 grid, 5 to 35 boxes	1.20 to 1.97
		test (hard)	60 by 60 to 100 by 100 grid, 4 to 80 boxes	1.40 to 2.02
	spanner	test (easy)	1 to 5 nuts, 4 to 10 locations	1.14 to 1.87
		test (medium)	15 to 50 nuts, 15 to 45 locations	1.11 to 3.24
		test (hard)	50 to 250 nuts, 50 to 100 locations	3.62 to 44.37
	transport	test (easy)	3 to 6 vehicles, 5 to 15 locations	5.00 to 79.56
		test (medium)	10 to 20 vehicles, 20 to 40 locations	27.35 to 612.02
		test (hard)	30 to 50 vehicles, 50 to 100 locations	143.04 to 1669.00
HBF	blocksworld-large	test (easy)	500 to 703 blocks, 1% in goal	102.05 to 159.17
		test (medium)	720 to 960 blocks, 0.9% in goal	124.50 to 161.33
		test (hard)	940 to 1221 blocks, 0.8% in goal	113.00 to 158.67
	transport-sparse	test (easy)	3 to 6 vehicles, 7 to 15 locations	3.71 to 13.76
		test (medium)	10 to 20 vehicles, 20 to 40 locations	10.15 to 68.31
		test (hard)	30 to 50 vehicles, 50 to 100 locations	71.39 to 195.62
	transport-dense	test (easy)	3 to 6 vehicles, 7 to 15 locations	7.00 to 39.79
		test (medium)	10 to 20 vehicles, 20 to 40 locations	105.65 to 551.28
		test (hard)	30 to 50 vehicles, 50 to 100 locations	854.42 to 1653.00
	transport-full	test (easy)	3 to 6 vehicles, 7 to 15 locations	14.80 to 53.20
		test (medium)	10 to 20 vehicles, 20 to 40 locations	180.21 to 687.43
		test (hard)	30 to 50 vehicles, 50 to 100 locations	1401.13 to 1879.50
	warehouse	test (easy)	10 to 100 boxes in 4 to 40 towers	7.33 to 1178.54
		test (medium)	100 to 150 boxes in 40 to 60 towers	1413.64 to 1962.33
		test (hard)	150 to 200 boxes in 60 to 80 towers	1623.00 to 2079.50

[a] In Childsnack test (hard) instances, the branching factor is so high that the planner failed to expand any state on any instance.

5.3 Experimental Methodology

We aim to empirically evaluate how well our contributions perform in practice. This ultimately means answering two questions: how well can they solve large and complex planning tasks, and how good are the plans they produce. For our learned action set heuristics, the first question is additionally requires us to understand how well they can generalise to tasks much larger than the training instances. To answer these questions, we will need to run experiments on our previously described benchmark sets.

5.3.1 Baselines

The baseline planning system we compare against are the current state-of-the-art in satisficing planning, LAMA-first, the current state-of-the-art in learning for planning, GOOSE, and the current state-of-the-art in lifted planning, Powerlifted. The details of these systems are as follows:

- As we had discussed in Section 2.1.2, LAMA-first is a variant of the LAMA planner that returns the first plan found. It outperformed all other planners in the IPC 2023 agile track.
- GOOSE is the learning for planning system that exploits the methods we had described in Section 2.3. GOOSE allows the use of multiple possible underlying planning systems, including Fast Downward and Powerlifted, and supports both classical planning and numeric planning[8]. We had also implemented the key classical planning component of GOOSE in LazyLifted. We expect our implementation to perform on par with or better than the original GOOSE implementation. Therefore, we use our own implementation of GOOSE as a fair baseline.

  Beyond having multiple implementations, GOOSE also has many possible configurations that control the hyperparameters and training method used. We use the recommended configuration from the GOOSE repository[9] to evaluate against the best version of GOOSE. Specifically, we train a ranking heuristic using three iterations of the WL algorithm with the Instance Learning Graph and the Linear Program formulation of the ranking problem with regularisation hyperparameter $C = 1$. This is exactly what we had described in Section 2.3.3.
- Powerlifted is a lifted planner that is based on the FF heuristic with additional heuristic search techniques. Like with GOOSE, we use the recommended configuration from the Powerlifted repository[10] to evaluate against the best version of Powerlifted.

By using the best versions of these state-of-the-art planners, we aim to provide a fair

[8] Numeric planning is an extension of classical planning with numeric state variables that actions can depend on and change.
[9] Available at https://github.com/DillonZChen/goose
[10] Available at https://github.com/abcorrea/powerlifted.

comparison of our contributions against the best that is currently available. This is important as it allows us to understand how much our contributions improve over the current state-of-the-art.

5.3.2 Training and Testing Instances

As mentioned in previous sections, our benchmark set contains training instances that are roughly on the same scale as the easy test instances. This is to allow us to understand if both GOOSE and our learning systems are at least able to learn how to solve similar tasks to the easy test instances. The medium and hard test instances then allow us to examine if they are able to generalise to more difficult tasks.

Both GOOSE and our learning systems use training plan traces that demonstrate how the training instances can be solved. For the IPC 2023 Learning Track domains, we use the same training plan traces as used by GOOSE in Chen, Trevizan, and Thiébaux (2024). For the high branching factor domains, we use the same methodology as used in Chen, Trevizan, and Thiébaux (2024) to generate training plan traces. Specifically, we run the Scorpion optimal planner on the training instances and use the resulting optimal plans as training plan traces. For both the Blocksworld Large and Warehouse domains, the Scorpion planner solves none or very few training instances. In these cases, we use LAMA-first to generate the training plan traces.

5.3.3 Hardware and Software

All of our experiments are run on a cluster with Intel Xeon 3.2 GHz CPU cores. When training heuristics for GOOSE and our learned action set heuristics, we use a single core with 32 GB of memory with a time limit of 24 hours. When running a planning system on a test instance, we use a single core with 8 GB of memory with a time limit of 30 minutes. These settings are chosen to be consistent with various existing works such as Chen, Trevizan, and Thiébaux (2024) and the IPC 2023 Learning Track (Taitler et al., 2024).

We examine a number of combinations of our contributions. For search space, we examine both state space search and partial space search. For the choice of the action set heuristics, we examine both the action set version of the FF heuristic and the learned action heuristics from our two graph representations, namely AOAG and AEG. For each graph representation, we train two action set heuristics, one using regression and one using ranking. Altogether, this results in five choices of action set heuristics. Like we had discussed, action set heuristics can be used for both state space search and partial space search. This means that we examine a total of ten combinations of our contributions[11]. Such an extensive examination is necessary to cover the full range of possibilities and understand how well our contributions perform in practice.

[11] Note that the FF heuristic with state space search is not actually our contribution, but we still examine it, mainly to observe how the FF heuristic performs with partial space search when compared with state space search.

To name these combinations, we use the following notation: `<search space>-<graph representation>-<heuristic type>`. Search space is either PS^2 for partial space search, or S^3 for state space search. For the FF heuristic, we omit the graph representation and use `FF` as the heuristic type. For learned action set heuristics, the graph representation is either `AOAG` or `AEG`, and the heuristic type is either `GPR` or `LP`. Here, `GPR` refers to regression using Gaussian Process Regressor and `LP` refers to ranking using our linear program formulation. We will now refer to each of these as a LazyLifted *configuration*.

Our learned action set heuristics additionally have a number of hyperparameters. Specifically, for regression based heuristics, we use four iterations of the WL algorithm to generate feature vectors and Gaussian process regression with a linear kernel and an α constant of 10^{-7}. For ranking based heuristics, we use two iterations of the WL algorithm to generate feature vectors and generate the training data set using importance parameters of $\sigma_{\text{lp}} = 0.5$, $\sigma_{\text{ls}} = 2.0$, $\sigma_{\text{sp}} = 0.5$, and $\sigma_{\text{ss}} = 1.0$. We use the Rank2Plan library to efficiently and automatically tune the regularisation hyperparameter C for the RankSVM in our ranking based heuristics, resulting in domain-specific auto-tuned C values. During this tuning process, we hold largest 20% of training instances as a validation set, and use the remaining 80% for training. Once the hyperparameters are tuned, we train the final ranking heuristic on the full training set.

For the warehouse domain, the high branching factor even on the training set resulted in an overwhelming number of layer and state sibling tuples in our ranking datasets. To alleviate this, our importance hyperparameters for Warehouse are $\sigma_{\text{lp}} = 2.0$, $\sigma_{\text{ls}} = 1.5$, $\sigma_{\text{sp}} = 2.0$, and $\sigma_{\text{ss}} = 0.5$.

The above hyperparameters are all chosen based on results of preliminary experiments. We do not perform a full hyperparameter search for these hyperparameters, as this would be computationally expensive. However, we believe that the hyperparameters chosen are reasonable and provide a good starting point for our experiments.

5.3.4 Metrics

When evaluating the performance of various planning systems, we are mainly concerned with two metrics: coverage and plan quality. Coverage is the number of tasks solved by a planning system within our 30 minutes time limit. It describes how effective a planning system is at solving tasks. Plan quality is also important to us, as one would clearly prefer better plans. In our case, plan quality is particularly interesting as our training plans are optimal except on the Blocksworld Large and Warehouse domains. We would then hope that the learning for planning systems (i.e., GOOSE and our learned action set heuristics) are able to produce plans that are close to optimal.

Coverage is trivial to measure, but to measure plan quality we need a metric. We base our metric on that used in the IPC 2023 Learning Track. There, they obtained reference plans for each task in the test set and measured the plan quality of the plans produced by the planning systems against these reference plans. Specifically, they measured the

quality score of a plan as C_{ref}/C, where C is the cost of the plan produced by the planning system and C_{ref} is the cost of the reference plan. The cost of a planner is then the sum of the quality scores of the plans it produces. In our case, we use the best plan found by any of the planning systems we examine as the reference plan. This is a reasonable choice, allowing us to compare the quality of the plans found by all the planning systems we examine. We call the resulting metric the quality score.

5.4 Results

In this section we present and analyse our experimental results. Before doing so, we first discuss in Section 5.4.1 the key questions we seek to answer using these results and our anticipated outcomes. We then present the results in Section 5.4.2 and analyse them in light of these questions, examining if reality matches our expectations. We will also discuss some additional questions and insights that arise from the results. Lastly in Section 5.4.3, we will discuss our results in detail in a domain by domain basis.

5.4.1 Key questions and anticipations

The most important question we seek to answer is how well our contributions perform, as measured by coverage and plan quality. This leads to a few more specific questions regarding each of our individual contributions:

- *How well does partial space search perform compared to state space search when using the same underlying heuristic (e.g., the FF heuristic)?* We anticipate that partial space search is likely not beneficial on tasks with low branching factors, such as Sokoban. Partial space search is designed to factorise the branching factor of planning tasks, and on low branching factor tasks this only introduces unnecessary overhead. However, this overhead should not be significant, and we do not expect partial space search to perform significantly worse than state space search on low branching factor tasks. On the other hand, we anticipate that partial space search will perform significantly better than state space search on tasks with high branching factors. This is because partial space search is exactly designed to handle high branching factor tasks, and we expect it to be able to do so effectively.
- *How good is automatic translation from state space heuristics to action set heuristics, as exemplified by the FF heuristic?* We do not have clear anticipations for this question. However, if the translation is good, we anticipate that the action set version will mostly reflect the performance characteristics of partial space search when compared with the original version using state space search. This is because a good translation means the heuristic preserves the same accuracy, in which case the only difference is the search space.
- *How good are the learned action set heuristics?* To address this question requires us to examine the performance of the learned action set heuristics using state space search, thereby removing the effect of partial space search. We anticipate that the

learned action set heuristics will be highly effective even with partial space search. This is because they are based on the same methodology as the ILG heuristics used by GOOSE, which is the current state-of-the-art in learning for planning. We also somewhat anticipate that the learned action set to outperform GOOSE, as they are trained using a larger training dataset thanks for partial space search.

- *How good are the plans found by learned action set heuristics?* This is particularly interesting, as the training plans for the learned action set heuristics are optimal except on the Blocksworld Large and Warehouse domains. We anticipate that the learned action set heuristics will also produce high quality plans thanks to their training on optimal plans.
- *How does partial space search affect plan quality?* We merely propose this as a key question, as we do not have clear anticipations for it. This is an important question because partial space search breaks down the search process into smaller steps, and it is not clear how this affects the quality of the plans produced.

5.4.2 Overview

We present the coverage and quality score of the planning systems we examine in Tables 5.2 and 5.3, respectively. These tables provide a comprehensive view of how well our contributions perform in comparison to the state-of-the-art baseline planners, and ultimately answer the key questions we had just posed.

How well do our contributions perform compared to the state-of-the-art baseline planners? This is the most important question we seek to answer. We can see from Table 5.2 and Table 5.3 that our learned action set heuristics trained through ranking are the best performing configurations amongst all our configurations, and we focus on them in our analysis.

In terms of coverage, we see that on the IPC 23 Learning Track domains, our learned action set heuristics with state space search outperform all systems except for LAMA-first. This is a significant improvement over the existing state-of-the-art in learning for planning, GOOSE, and in lifted planning, Powerlifted. This is a good indication of the performance of our systems. On the high branching factor domains, our learned action set heuristics with partial space search perform better than all baselines. This highlights the synergy of partial space search and learned action set heuristics, as we had anticipated. Altogether, PS^2-AOAG-LP is the best performing system. It outperforms GOOSE by around 25% and LAMA-first by around 5%. This is a landmark achievement, as it is the first learning for planning system to outperform LAMA-first.

In terms of quality score, similar trends continue. Our learned action set heuristics perform well on the IPC23 LT set with state space search and on the high branching factor domains with partial space search. The quality score of S^3-AEG-LP outperforms all baselines on the IPC 23 Learning Track domains, and the quality score of PS^2-AOAG-LP outperforms all baselines on the high branching factor domains. This is a

Table 5.2: Coverage of various planning systems. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Set	Domain	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
IPC23 LT	blocksworld	60	86	37	29	**90**	75	89	74	24	84	42	89	46
	childsnack	35	37	0	16	**38**	15	37	16	14	15	9	9	13
	ferry	68	**90**	0	60	88	75	**90**	81	49	87	69	**90**	58
	floortile	**11**	2	9	10	1	2	1	2	6	1	1	1	1
	miconic	**90**	**90**	81	77	**90**	**90**	**90**	**90**	68	**90**	**90**	**90**	**90**
	rovers	**69**	40	53	28	32	32	34	27	34	26	31	34	27
	satellite	**89**	39	0	49	53	47	39	12	50	41	57	29	24
	sokoban	**40**	27	34	32	27	29	27	29	29	24	26	27	26
	spanner	30	71	30	30	71	68	71	**73**	30	72	68	72	60
	transport	**66**	35	53	36	50	28	53	21	38	54	34	54	31
sum IPC coverage		**558**	517	297	367	540	461	531	425	342	494	427	495	376
HBF	blocksworld-large	7	40	2	0	18	0	35	0	0	48	0	**74**	0
	transport-sparse	62	30	58	31	37	14	41	21	31	58	30	**64**	31
	transport-dense	**66**	31	52	39	51	31	59	27	36	57	33	57	30
	transport-full	**66**	33	0	42	61	42	63	37	41	60	52	55	32
	warehouse	30	15	**90**	35	27	88	58	88	54	49	79	79	58
sum HBF coverage		231	149	202	147	194	175	256	173	162	272	194	**329**	151
sum total coverage		789	666	499	514	734	636	787	598	504	766	621	**824**	527

Table 5.3: Quality score of various planning systems rounded to integers. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Set	Domain	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
IPC23 LT	blocksworld	36	83	21	13	**89**	65	88	64	12	82	35	88	40
	childsnack	22	37	0	12	**38**	15	37	16	8	14	9	9	11
	ferry	57	87	0	55	86	71	89	79	33	86	67	**90**	53
	floortile	**10**	2	9	9	1	2	1	2	4	1	1	0	1
	miconic	73	89	77	76	89	89	89	**89**	55	81	84	82	84
	rovers	**66**	23	51	25	17	17	18	12	30	13	15	16	12
	satellite	**86**	28	0	47	41	26	26	3	43	25	33	17	9
	sokoban	**35**	19	28	26	19	23	19	23	23	11	18	14	17
	spanner	30	62	30	30	60	59	60	**70**	30	61	59	61	55
	transport	**63**	30	46	29	43	16	42	14	26	43	25	41	22
sum IPC quality		477	459	262	322	**482**	382	469	372	264	419	347	418	303
HBF	blocksworld-large	7	32	2	0	11	0	25	0	0	37	0	**60**	0
	transport-sparse	**59**	24	48	20	23	6	27	13	21	38	18	47	18
	transport-dense	**62**	27	45	30	46	19	49	17	24	36	27	46	24
	transport-full	**63**	30	0	38	56	33	51	29	32	40	46	41	29
	warehouse	30	6	**89**	35	14	82	54	82	54	17	68	57	40
sum HBF quality		220	118	184	124	150	141	205	141	131	168	160	**251**	111
sum total quality		**698**	578	446	445	632	523	675	513	394	587	506	669	414

good indication that our learned action set heuristics are able to produce high quality plans, as we had anticipated.

Are our methods outperforming the state-of-the-art baselines only because of our added high branching factor benchmarks? No, this is not the case. We can see that our learned action set heuristics are highly competitive with the LAMA-first on the IPC 23 Learning Track domains in terms of coverage and even outperform it in terms of quality score. Moreover, we outperform GOOSE on both metrics on the IPC set and vastly outperform Powerlifted.

Our added high branching factor benchmarks are also not a bias towards our own methods. Instead, existing benchmarks from the IPC 23 Learning Track are tasks that baselines like LAMA-first can solve reasonably well. That is, they are potentially biased towards the baselines. Our added benchmarks simply help us explore outside the capabilities of the baselines. As we discussed when introducing these benchmarks, they each explore slightly different capabilities. Blocksworld-large explores the ability to deal with irrelevant information, the transport variants explore the impact of graph density on branching factor, and warehouse explores the ability to deal with extreme high branching factor tasks. Our results show that our learned action set heuristics are able to handle all of these tasks effectively.

How well does partial space search perform compared to state space search when using the same underlying heuristic? To answer this question we need to compare the LazyLifted configurations using state space search (in the middle of the tables) to the configurations using partial space search (to the right of the tables). We can see that partial space search is generally not beneficial on the IPC 23 Learning Track domains. This is what we had anticipated — the low branching factor of these domains means that partial space search provides limited benefit while introducing unnecessary overhead. This trend holds for both coverage and quality, with a general decrease in both of these metrics by 10 to 20 percentage points. On the other hand, partial space search is generally beneficial on high branching factor tasks, offering improvements both in coverage and quality. This matches exactly what we had anticipated. Overall, we can conclude that partial space search achieves its design goal of improving performance on high branching factor tasks while maintaining reasonable performance on low branching factor tasks.

It is also interesting to note that the performance of partial space search when compared with state space search is dependent on the underlying heuristic. Specifically, stronger heuristics like AOAG-LP and AEG-LP work better with partial space search than weaker heuristics. That is, their performance when using partial space search is closer to their performance when using state space search for IPC 23 Learning Track domains, and they benefit more on high branching factor tasks. This also matches with our discussions in Section 3.4, where we discussed that partial space search moves the speed versus informedness trade-off towards informedness, and therefore works better

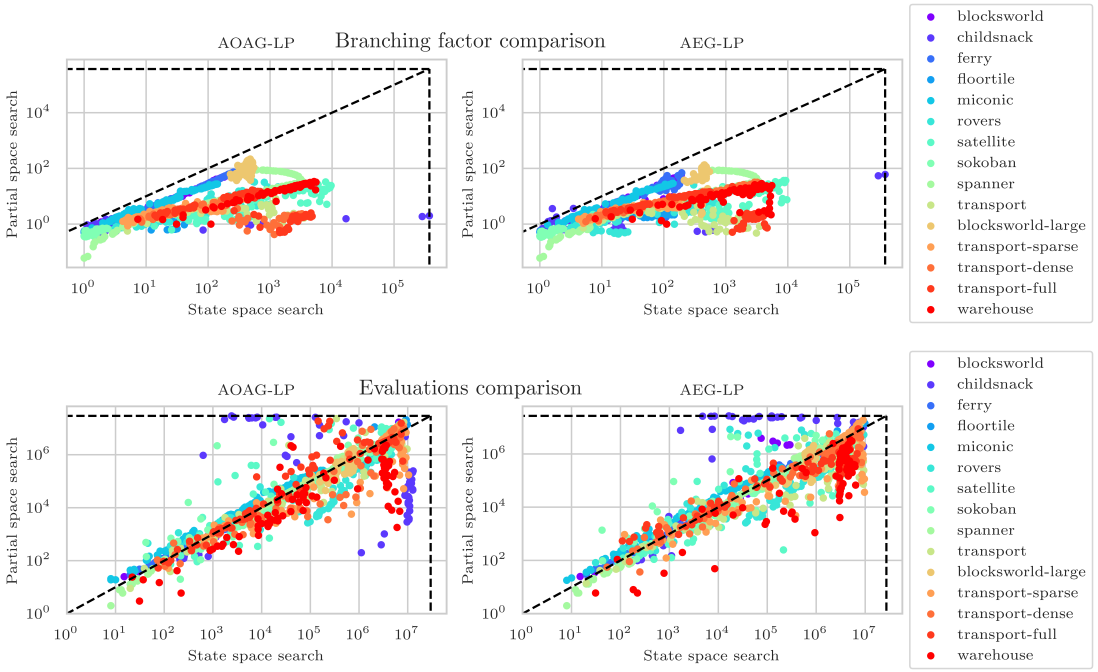


Figure 5.7: Comparison of branching factor (top) and number of nodes evaluated (bottom) for partial space search versus state space search, under our best two heuristics (AOAG-LP to the left and AEG-LP to the right). The x-axis is for state space search and y axis is for partial space search. The $x = y$ line is shown in the diagonal. Points below it favour partial space search, points above it favour state space search.

with more informed heuristics. This is a good indication that partial space search works as intended.

To provide more insight, we also directly examine the number of nodes evaluated during the search process and the branching factor of the tasks, when using partial space search versus state space search. We examine these for our best two heuristics, AOAG-LP and AEG-LP, in Figure 5.7. These plots show that partial space search produces a significant reduction in branching factor (nodes evaluated divided by nodes expanded) over state space search. However, partial space search also requires more node expansions, since it breaks a single search step into multiple steps. This is why the advantage is less clear when examining only nodes evaluated, and partial space search even leads to more evaluations on some domains. However, on most domains, particularly those with high branching (warm colours in the plot) factor, partial space search is able to reduce the number of evaluations, thereby saving search effort.

How good is the automatic translation from state space heuristics to action set heuristics, as exemplified by the FF heuristic? We answer this question by comparing the S^3-FF configuration against the PS^2-FF configuration. This allows us to examine how well the FF heuristic preserves its performance when translated to an action set heuristic. Again, the results match our anticipations. The FF heuristic performs slightly worse with PS^2 than with S^3 on IPC 23 Learning Track domains, but the difference is not significant. As we had discussed, we attribute much of this reduction in performance to partial space search, not to the translation process. On the other hand, the FF heuristic performs slightly better with PS^2 than with S^3 on high branching factor tasks. This is a good indication that our translation process is effective, as the FF heuristic is able to take advantage of partial space search to improve its performance on high branching tasks through the translation process.

How good are the learned action set heuristics? As we had discussed, our learned action set heuristics are the best performing configurations amongst all our configurations. In fact, when used with state space search, they outperform the state-of-the-art learned heuristic by GOOSE, which they are based on. This is an indication that we had learned stronger heuristics than the state-of-the-art in learning for planning. This is particularly interesting, as these heuristics are not exposed to action related features when performing inference in state space search, and instead they use essentially the same graph representations as GOOSE. As such, we view this as indication that our training process is effective. Specifically, this suggests that training even state space heuristics as action set heuristics with partial space search is beneficial, possibly due to the larger training dataset that partial space search provides, as shown in Table 5.4.

How good are the plans found by learned action set heuristics? Since the quality score is a summarisation of both coverage and the quality of the individual plans, we can observe the latter by examining quality scores in context of coverage. Under this analysis, we observe that our learned action set heuristics produce high quality plans especially when used with state space search. For example, S^3-AEG-LP has the highest quality score on the IPC set, despite having a slightly lower coverage than LAMA-first. This indicates that it finds high quality plans when it is able to solve tasks.

However, our learned action set heuristics do not seem to produce high quality plans specifically for transport and its variants. This is likely due to the path finding nature of these domains. Our learned heuristics are based on the WL algorithm, which cannot reason well about path finding as the WL algorithm focuses on the local information within graph representations. This is a limitation of WL based heuristics in general, not a limitation of our learned heuristics specifically — GOOSE also exhibits this limitation as it is based on the same methodology.

How does partial space search affect plan quality? We again answer this question by viewing the quality scores in context of coverage. This time we compare the plan

Table 5.4: Training data size for each domain when generating ranking datasets using state space search (S^3) versus partial space search (PS^2), along with the ratio of partial space search to state space search.

Domain	S^3	PS^2	PS^2/S^3
blocksworld	6237	14525	2.33
childsnack	9687	18877	1.95
ferry	11882	22036	1.85
floortile	28943	116668	4.03
miconic	21404	35217	1.65
rovers	18673	64685	3.46
satellite	59999	145636	2.43
sokoban	7657	37584	4.91
spanner	3283	14057	4.28
transport	13509	33304	2.47
blocksworld-large	118882	164968	1.39
transport-sparse	9865	32733	3.32
transport-dense	21577	47918	2.22
transport-full	36970	66932	1.81
warehouse	192665	237826	1.23

qualities of the same heuristics with state space search versus partial space search. We observe that partial space search generally seems to lead to slightly lower quality plans. This is likely a result of the fact that partial space search breaks down the search process into smaller steps. We hypothesise that it is harder for planners to consistently make good decisions when the search process is broken down into smaller steps, and small mistakes can accumulate over time into lower quality plans.

How much computational resource do our action set heuristics take to train? Thanks to the Rank2Plan library we had developed, most of our action set heuristics take a few minutes and 1-2 GB of memory to train. The most computationally expensive models take up to a few hours and 10-20 GB of memory to train. This is a very efficient training process, especially when compared to modern deep learning models that can take days to train.

5.4.3 Per domain analysis

Here we analyse our results in more detail on a domain by domain basis. We examine the coverage and quality score of the planning systems we examine on each domain, and discuss the key insights that arise from these results.

Table 5.5: Results for the blocksworld domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Metric	Test Set	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	**30**	**30**	29	**30**	**30**	**30**	29	24	**30**	**30**	**30**	**30**
	Medium	27	29	7	0	**30**	27	29	29	0	**30**	11	29	15
	Hard	3	27	0	0	**30**	18	**30**	16	0	24	1	**30**	1
	Total	60	86	37	29	**90**	75	89	74	24	84	42	89	46
Quality	Easy	19.7	28.8	17.8	13.0	29.5	27.4	29.4	26.4	11.8	29.3	26.6	**29.6**	27.1
	Medium	14.8	27.9	3.4	0.0	**29.5**	22.9	28.5	24.9	0.0	29.2	7.8	28.8	11.9
	Hard	1.2	26.0	0.0	0.0	29.7	14.6	29.7	12.8	0.0	23.6	0.8	**29.8**	0.8
	Total	35.6	82.7	21.2	13.0	**88.6**	64.9	87.6	64.2	11.8	82.2	35.2	88.1	39.8

Table 5.6: Results for the blocksworld-large domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Metric	Test Set	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	7	29	2	0	15	0	29	0	0	24	0	**30**	0
	Medium	0	10	0	0	3	0	6	0	0	11	0	**25**	0
	Hard	0	1	0	0	0	0	0	0	0	13	0	**19**	0
	Total	7	40	2	0	18	0	35	0	0	48	0	**74**	0
Quality	Easy	7.0	**21.0**	1.9	0.0	9.0	0.0	19.1	0.0	0.0	14.9	0.0	18.2	0.0
	Medium	0.0	9.6	0.0	0.0	2.5	0.0	5.5	0.0	0.0	10.2	0.0	**23.7**	0.0
	Hard	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	12.2	0.0	**18.0**	0.0
	Total	7.0	31.6	1.9	0.0	11.5	0.0	24.6	0.0	0.0	37.3	0.0	**59.8**	0.0

Blocksworld and Blocksworld Large

We discuss Blocksworld and Blocksworld Large together due to their inherent connection. An overview of their results is shown in Table 5.5 and 5.6. It is clear that in terms of coverage, our action set heuristics learned from ranking (AOAG-LP and AEG-LP) lead to the best performing systems, whether using state space search or partial space search. This is reflected in terms of both coverage and quality score. This is a good indication that our systems are able to both solve tasks efficiently and product high quality plans in the same time.

It is also interesting to observe the difference in planner performances between the Blocksworld and Blocksworld Large domains. On Blocksworld, state space search and partial space search perform similarly, while on Blocksworld Large, partial space search is significantly better. This corresponds exactly to the capability of partial space search to handle high branching factor tasks. Moreover, it is also interesting to observe that high performing systems on Blocksworld Large tend to produce low quality plans, judging by the drop in value from coverage to quality score. It is interesting to investigate the cause of this, given that our systems tend to produce the best plans on Blocksworld. We do not believe this is due to using the satisficing LAMA-first instead of the optimal Scorpion as the training plan generator for Blocksworld Large, as it produces very high quality plans for the instances it does solve.

We can make further observations through the planning time results from our experimental logs. Looking at Blocksworld, it is clear that our methods are not able to solve small instances as quickly as baselines, however, the time they need to solve problems increases much slower than the baselines as the problem size increases. This is what allows them to ultimately outperform the baselines. On the other hand, on Blocksworld Large, our methods are the only ones able to solve many instances and hence obviously have the shortest planning times. However, the planning time fluctuates significantly between instances, suggesting unstable performance. Nonetheless, they are the only systems capable of solving many instances, which is important in its own right.

Childsnack

An overview of the results for Childsnack is shown in Table 5.7. It is worth noting that Powerlifted (PWL) cannot solve any instances because it cannot deal with the negative action schema preconditions in this domain. These results indicate that our strongest baseline is GOOSE, and that our systems produce similar results in terms of both coverage and quality score to GOOSE. This is unsurprising, given that our methods are based on those used in GOOSE. Nonetheless, our systems are able to outperform GOOSE slightly in terms of both metrics. The detailed plan costs results from our logs validate again that our systems and GOOSE produce similar high quality plans. However, the planning time results from our logs show that both GOOSE and our systems are noticeably slower than LAMA-first in solving Childsnack instances, even though we solve more instances overall. This suggests that our methods are more robust,

Table 5.7: Results for the childsnack domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Metric	Test Set	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	29	**30**	0	16	**30**	15	**30**	16	14	15	9	9	13
	Medium	5	7	0	0	**8**	0	6	0	0	0	0	0	0
	Hard	**1**	0	0	0	0	0	**1**	0	0	0	0	0	0
	Total	35	37	0	16	**38**	15	37	16	14	15	9	9	13
Quality	Easy	18.7	**29.9**	0.0	11.9	29.8	14.7	**29.9**	16.0	8.2	14.4	8.6	8.9	11.0
	Medium	3.0	7.0	0.0	0.0	**8.0**	0.0	6.0	0.0	0.0	0.0	0.0	0.0	0.0
	Hard	0.6	0.0	0.0	0.0	0.0	0.0	**1.0**	0.0	0.0	0.0	0.0	0.0	0.0
	Total	22.3	36.9	0.0	11.9	**37.8**	14.7	36.9	16.0	8.2	14.4	8.6	8.9	11.0

but less agile, than LAMA-first.

Another important observation on Childsnack is that partial space search performs significantly worse than state space search, even when using the same heuristic. We hypothesise this is a consequence of shortsightedness caused by partial space search. Specifically, by partial space search breaking down each search step into multiple smaller steps, the search process looks a smaller amount ahead when making decisions. This is particularly detrimental in Childsnack, possibly due to the high potential for reaching deadends if one does not look far enough. This suggests that partial space search is not always better than state space search, and that it is important to consider the nature of the task when choosing between the two. Nonetheless, it is worth noting that Childsnack is the only domain where partial space search performs significantly worse than state space search. Perhaps dedicated dead-end avoiding methods can help alleviate the problem and make partial space search generally stronger than state space search.

Ferry

An overview of the results for Ferry is shown in Table 5.8. Again, Powerlifted (PWL) cannot solve anything because it cannot deal with negative preconditions. These results indicate the strength of our system on Ferry, both in terms of coverage and quality. It is worth noting that the original GOOSE implementation does not perform as well as we present on Ferry. Our implementation of GOOSE is lifted, allowing it to handle much larger tasks that the original implementation would have struggled with. Nonetheless, our systems are able to solve all tasks like GOOSE, and produce slightly better plans

Table 5.8: Results for the ferry domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Metric	Test Set	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	**30**	0	**30**	**30**	**30**	**30**	29	**30**	**30**	**30**	**30**	**30**
	Medium	**30**	**30**	0	**30**	**30**	**30**	**30**	**30**	19	**30**	**30**	**30**	28
	Hard	8	**30**	0	0	28	15	**30**	22	0	27	9	**30**	0
	Total	68	**90**	0	60	88	75	**90**	81	49	87	69	**90**	58
Quality	Easy	25.4	29.5	0.0	28.3	29.6	28.7	29.6	28.4	21.0	29.7	29.1	**29.8**	29.3
	Medium	24.9	28.9	0.0	26.9	29.2	28.7	29.7	29.2	11.7	29.4	29.5	**29.9**	23.8
	Hard	6.8	28.3	0.0	0.0	27.2	14.0	29.8	21.1	0.0	26.5	8.7	**30.0**	0.0
	Total	57.2	86.7	0.0	55.2	86.0	71.4	89.2	78.7	32.7	85.6	67.4	**89.8**	53.1

according to plan cost results.

Looking at the planning time results in our logs, we see that our systems repeat the trend from Blocksworld. That is, we are not as quick as LAMA-first to solve small instances, but our planning time scales much better with problem size. This is again what allows us to outperform LAMA-first. Moreover, our systems produce much better plans than LAMA-first.

Floortile

An overview of the results for Floortile is shown in Table 5.9. In general, all systems struggle with Floortile. In particular, the learning based systems (GOOSE and our systems that are not FF) struggle more with Floortile. This suggests that our weakness comes from the fundamental WL based methodology that we and GOOSE use.

Miconic

An overview of the results for Miconic is shown in Table 5.10. These results largely reflect the same patterns as the results on Blocksworld and Ferry, except that Miconic appears to be a much easier domain for all systems. Most systems solve all tasks, and most learning based systems produce high quality plans. The plan cost results in our logs confirm this. It is worth noting though that partial space search seems to lead to slightly worse plans, though the difference is not significant. The planning time results show that LAMA-first is much quicker than other systems to solve Miconic tasks. This

Table 5.9: Results for the floortile domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

Metric	Test Set	Baseline				New								
		LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**11**	2	9	10	1	2	1	2	6	1	1	1	1
	Medium	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	Hard	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	Total	**11**	2	9	10	1	2	1	2	6	1	1	1	1
Quality	Easy	**10.2**	1.9	8.9	9.0	0.6	1.6	0.6	1.7	4.3	0.6	0.8	0.4	0.8
	Medium	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
	Hard	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
	Total	**10.2**	1.9	8.9	9.0	0.6	1.6	0.6	1.7	4.3	0.6	0.8	0.4	0.8

Table 5.10: Results for the miconic domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

Metric	Test Set	Baseline				New								
		LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
	Medium	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
	Hard	**30**	**30**	21	17	**30**	**30**	**30**	**30**	8	**30**	**30**	**30**	**30**
	Total	**90**	**90**	81	77	**90**	**90**	**90**	**90**	68	**90**	**90**	**90**	**90**
Quality	Easy	25.7	29.4	29.3	**29.5**	29.4	29.5	29.4	29.5	25.9	27.0	27.3	26.6	27.2
	Medium	23.4	**29.8**	28.2	29.5	**29.8**	29.7	**29.8**	29.8	23.3	27.0	28.5	27.3	28.6
	Hard	23.6	29.8	19.2	16.7	29.8	29.5	29.8	**30.0**	6.3	27.4	27.9	27.8	28.3
	Total	72.7	89.1	76.7	75.8	89.1	88.7	89.1	**89.2**	55.5	81.4	83.8	81.7	84.1

Table 5.11: Results for the rovers domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Metric	Test Set	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	**30**	**30**	27	**30**	**30**	**30**	27	**30**	26	**30**	**30**	27
	Medium	**29**	10	22	1	2	2	4	0	4	0	1	4	0
	Hard	**10**	0	1	0	0	0	0	0	0	0	0	0	0
	Total	**69**	40	53	28	32	32	34	27	34	26	31	34	27
Quality	Easy	28.4	18.7	**28.6**	23.9	16.1	15.7	16.6	11.9	26.5	13.2	15.0	14.7	11.7
	Medium	**28.1**	4.7	21.7	0.8	0.8	1.4	1.7	0.0	3.2	0.0	0.4	1.6	0.0
	Hard	**9.9**	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Total	**66.4**	23.3	51.3	24.7	16.9	17.1	18.3	11.9	29.7	13.2	15.4	16.3	11.7

suggests that our systems offer a different trade-off between speed and quality than LAMA-first.

Rovers

An overview of the results for Rovers is shown in Table 5.11. These results indicate that our systems are generally weaker than our baselines on Rovers. This likely suggests a weakness in our methodology that is worth future investigation. The plan cost results in our logs show that when we do find a plan, it is of reasonably high quality. However, the planning time results in our logs give the same indication as coverage, that our systems are generally weaker than our baselines on Rovers.

Satellite

An overview of the results for Satellite is shown in Table 5.12. Powerlifted does not solve anything as it cannot deal with the negative preconditions in this domain. These results indicate that our systems are again weaker than LAMA-first on Satellite but perform better than GOOSE. The plan cost results in our logs show that the quality of the plans we produce are also weaker than LAMA-first. Moreover, the planning time results from our logs show that when we do find a plan, we usually do so slower than LAMA-first. These results altogether indicate that our systems are weak on Satellite. However, on the bright side, we improve upon the results from GOOSE on Satellite. This suggests that our methodologies are actually beneficial on GOOSE, but the WL based methodology we inherit from GOOSE is fundamentally weak on Satellite.

Table 5.12: Results for the satellite domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Metric	Test Set	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	27	0	**30**	**30**	**30**	29	12	**30**	**30**	**30**	28	24
	Medium	**30**	12	0	19	23	17	10	0	20	11	27	1	0
	Hard	**29**	0	0	0	0	0	0	0	0	0	0	0	0
	Total	**89**	39	0	49	53	47	39	12	50	41	57	29	24
Quality	Easy	28.1	18.7	0.0	**28.3**	22.7	16.7	20.7	3.3	26.5	19.4	18.6	16.4	8.5
	Medium	**28.4**	9.7	0.0	18.6	18.3	9.2	5.8	0.0	16.3	5.8	14.4	0.5	0.0
	Hard	**29.0**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Total	**85.5**	28.4	0.0	46.9	41.0	26.0	26.5	3.3	42.8	25.2	33.0	17.0	8.5

Sokoban

An overview of the results for Sokoban is shown in Table 5.13. Like with Satellite, these results show that our systems are weaker than our baselines, namely LAMA-first and Powerlifted. However, the weakness is not as strong as on Satellite, and we slightly outperform GOOSE. The plan cost results in our logs show that we are able to find reasonably competitive plans when using state space search, but plan quality suffers under partial space search. The planning time results show that our systems have a very high variance in planning time. However, all of these properties are shared with GOOSE, suggesting that our contributions do not harm performance on GOOSE, and the weaknesses are inherent to the WL based methodology we inherit from GOOSE.

Spanner

An overview of the results for Spanner is shown in Table 5.14. These results indicate that GOOSE and our learning based systems are much stronger than the other systems. This suggests, opposite to Satellite and Sokoban, that the WL based methodology we inherit from GOOSE is very strong on Spanner. The plan cost results in our logs show that our systems are able to produce betters plans than GOOSE. The planning time results further show that our systems are able to solve Spanner tasks slightly quicker than GOOSE. Altogether, these results show that our systems are able to improve upon the state-of-the-art in learning for planning on Spanner.

Table 5.13: Results for the sokoban domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

Metric	Test Set	Baseline				New								
		LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	27	**30**	**30**	27	29	27	29	28	24	26	27	26
	Medium	**10**	0	4	2	0	0	0	0	1	0	0	0	0
	Hard	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	Total	**40**	27	34	32	27	29	27	29	29	24	26	27	26
Quality	Easy	**25.4**	18.7	24.3	24.8	18.7	22.7	18.7	23.4	21.8	11.5	18.4	13.8	17.3
	Medium	**9.2**	0.0	3.7	1.4	0.0	0.0	0.0	0.0	0.8	0.0	0.0	0.0	0.0
	Hard	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
	Total	**34.6**	18.7	28.0	26.3	18.7	22.7	18.7	23.4	22.6	11.5	18.4	13.8	17.3

Table 5.14: Results for the spanner domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

Metric	Test Set	Baseline				New								
		LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
	Medium	0	**30**	0	0	**30**	**30**	**30**	**30**	0	**30**	**30**	**30**	29
	Hard	0	11	0	0	11	8	11	**13**	0	12	8	12	1
	Total	30	71	30	30	71	68	71	**73**	30	72	68	72	60
Quality	Easy	**30.0**	27.0	**30.0**	**30.0**	25.9	26.2	25.9	26.7	**30.0**	25.9	26.1	25.9	26.1
	Medium	0.0	26.2	0.0	0.0	25.7	25.9	25.7	**30.0**	0.0	25.7	26.4	25.7	27.7
	Hard	0.0	8.8	0.0	0.0	8.7	6.5	8.7	**13.0**	0.0	9.7	6.7	9.7	1.0
	Total	30.0	62.0	30.0	30.0	60.3	58.6	60.3	**69.7**	30.0	61.4	59.2	61.4	54.7

Table 5.15: Results for the transport domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

Metric	Test Set	Baseline				New								
		LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	**30**	**30**	**30**	**30**	28	**30**	21	**30**	**30**	29	**30**	**30**
	Medium	**30**	5	23	6	20	0	23	0	8	22	5	24	1
	Hard	**6**	0	0	0	0	0	0	0	0	2	0	0	0
	Total	**66**	35	53	36	50	28	53	21	38	54	34	54	31
Quality	Easy	**27.5**	25.7	27.0	23.9	26.0	16.4	24.8	13.9	20.6	24.6	22.6	24.5	21.8
	Medium	**29.4**	4.1	19.2	5.2	16.8	0.0	17.4	0.0	5.3	17.0	2.1	16.1	0.3
	Hard	**6.0**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.8	0.0	0.0	0.0
	Total	**62.9**	29.8	46.2	29.1	42.8	16.4	42.2	13.9	25.9	43.3	24.7	40.6	22.1

Transport and its variants

We analyse all four transport related domains together due to their inherent connection. Overviews of their results are shown in Table 5.15 to 5.18. These results show that our systems generally improve upon the results obtained by GOOSE on the transport related domains. This indicates that our contributions are effective on these domains. Moreover, our systems are able to achieve competitive results with LAMA-first on the transport related domains, indicating strong overall performance. However, as the plan cost results in our logs show, our systems are not able to produce plans of similar quality to LAMA-first on large instances. Moreover, the planning time results show that our systems are generally slightly slower than LAMA-first to solve transport related tasks. Nonetheless, our contributions improve upon GOOSE, which they are based on, on the transport domains.

It is also interesting to observe the impact of graph density on the effectiveness of planning systems on transport domains. We are particularly interested in comparing state space search and partial space search here, to see how density affects the effectiveness of partial space search. We can see that on transport sparse, partial space search is significantly better than state space search. As density increases, the advantage of partial space search diminishes. This is contrary to the naive expectation that partial space search would do better on denser graphs due to the higher branching factor. We hypothesise partial space search does well on sparser graphs due to the average distance that needs to be traversed being higher, as indicated by the reduction in plan costs for denser transport domains. This ultimately means that the high branching factor on

Table 5.16: Results for the transport-sparse domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

Metric	Test Set	Baseline				New								
		LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	**30**	**30**	29	**30**	14	**30**	21	29	**30**	27	**30**	**30**
	Medium	**30**	0	27	2	7	0	11	0	2	27	3	**30**	1
	Hard	2	0	1	0	0	0	0	0	0	1	0	**4**	0
	Total	62	30	58	31	37	14	41	21	31	58	30	**64**	31
Quality	Easy	**27.4**	24.0	26.7	18.7	18.9	6.1	18.5	12.5	19.4	20.6	17.3	20.7	17.1
	Medium	**29.3**	0.0	20.9	1.7	4.5	0.0	8.5	0.0	1.3	17.0	1.0	22.5	0.4
	Hard	2.0	0.0	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	**3.7**	0.0
	Total	**58.7**	24.0	48.3	20.4	23.4	6.1	27.0	12.5	20.7	38.1	18.3	46.8	17.5

Table 5.17: Results for the transport-dense domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

Metric	Test Set	Baseline				New								
		LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	**30**	**30**	**30**	**30**	**30**	**30**	27	**30**	**30**	**30**	**30**	**30**
	Medium	**30**	1	22	9	21	1	27	0	6	27	3	25	0
	Hard	**6**	0	0	0	0	0	2	0	0	0	0	2	0
	Total	**66**	31	52	39	51	31	59	27	36	57	33	57	30
Quality	Easy	**27.4**	25.7	26.5	23.1	26.0	18.9	26.3	16.7	20.4	21.0	25.1	25.7	24.3
	Medium	**28.7**	1.0	18.3	7.1	19.7	0.4	21.2	0.0	4.0	14.9	2.0	19.9	0.0
	Hard	**6.0**	0.0	0.0	0.0	0.0	0.0	1.4	0.0	0.0	0.0	0.0	0.9	0.0
	Total	**62.1**	26.7	44.8	30.2	45.8	19.3	48.9	16.7	24.3	36.0	27.1	46.4	24.3

Table 5.18: Results for the transport-full domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Metric	Test Set	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	**30**	0	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
	Medium	**30**	3	0	12	**30**	12	27	7	11	**30**	22	25	2
	Hard	**6**	0	0	0	1	0	**6**	0	0	0	0	0	0
	Total	**66**	33	0	42	61	42	63	37	41	60	52	55	32
Quality	Easy	27.2	27.6	0.0	27.1	26.4	25.0	25.0	24.8	22.8	17.8	**27.8**	24.2	27.5
	Medium	**29.3**	2.9	0.0	10.9	28.4	8.0	21.5	4.4	8.8	21.9	18.5	16.6	1.4
	Hard	**6.0**	0.0	0.0	0.0	0.9	0.0	4.7	0.0	0.0	0.0	0.0	0.0	0.0
	Total	**62.5**	30.5	0.0	38.0	55.6	33.0	51.1	29.2	31.7	39.7	46.3	40.9	28.9

denser graphs is paid less often, and the advantage of partial space search diminishes. On sparser graphs, the branching factor is still not low, and at the same time plans are longer, so the advantage of partial space search is more pronounced.

Warehouse

An overview of the results for Warehouse is shown in Table 5.19. These results indicate that our systems vastly outperform LAMA-first and GOOSE, and are competitive with Powerlifted on Warehouse. This indicates the overall strength of our systems. The plan cost results in our logs show that our systems are able to produce competitive plans on Warehouse. The planning time results show that our systems are generally slower than Powerlifted to solve Warehouse tasks. This indicates that there is still room for improvement in our systems on Warehouse, however they already mark a major step forward in learning for planning on this domain.

It is interesting to note that partial space search actually performs worse than state space search on Warehouse. This is not what one may anticipate due to the high branching factor on Warehouse. However, as the plan cost results indicate, almost all tasks on Warehouse can be solved in short plans, so the branching factor is actually not paid very often. This is likely why partial space search does not offer an advantage on Warehouse.

Table 5.19: Results for the warehouse domain. The best score for each row is highlighted in bold. The top three unique scores for each row are highlighted in different shades of green with darker being better.

		Baseline				New								
Metric	Test Set	LAMA-F	GOOSE	PWL	S^3-FF	S^3-AEG-LP	S^3-AEG-GPR	S^3-AOAG-LP	S^3-AOAG-GPR	PS^2-FF	PS^2-AEG-LP	PS^2-AEG-GPR	PS^2-AOAG-LP	PS^2-AOAG-GPR
Coverage	Easy	**30**	13	**30**	28	21	**30**	24	**30**	**30**	29	26	**30**	27
	Medium	0	2	**30**	7	6	**30**	18	**30**	24	18	26	29	15
	Hard	0	0	**30**	0	0	28	16	28	0	2	27	20	16
	Total	30	15	**90**	35	27	88	58	88	54	49	79	79	58
Quality	Easy	**30.0**	3.7	29.3	27.9	8.9	28.8	22.1	28.8	29.9	10.7	21.9	25.1	20.7
	Medium	0.0	2.0	**29.8**	7.0	5.0	27.6	17.0	27.8	23.9	5.5	19.9	18.8	10.5
	Hard	0.0	0.0	**29.4**	0.0	0.0	25.9	14.5	25.8	0.0	0.7	26.1	12.9	9.0
	Total	30.0	5.7	**88.6**	34.9	13.9	82.3	53.6	82.4	53.9	16.9	67.9	56.8	40.3

Chapter 6

Related Work

Our focus is in the field of learning for planning. With the success of deep learning techniques in the last decade, a large body of work has emerged in learning for planning. Nevertheless, to our knowledge, our work is still novel in that it is the first to consider how learning and planning components can be better integrated together, rather than just how learning can be used to help planning. As we had discussed in previous chapters, this is theoretically important and highly beneficial in practice.

To better describe the place and novelty of our work in literature, we provide a general overview of the field of learning for planning. We will categorise the field into four main areas: learning heuristics for planning, learning generalised policies for planning, other methods to aid planning with learning, and planning model design with learning. We will discuss these areas in the next four sections. Our work is most related to the first area as we learn heuristics for guiding partial space search.

Our contribution of partial space search is designed to take advantage of learning systems, but is not restricted to learning methods. More broadly, it is related with other works that aim to reduce the search space of planning tasks. We will discuss these works in the last section of this chapter.

6.1 Learning Heuristics for Planning

Learning heuristics for planning is perhaps the most researched area in learning for planning. To our knowledge, all existing works in this area learn state space heuristics, whereas as we introduced the notion of action set heuristics and learned action set heuristics. Still, state space heuristics and action set heuristics are deeply related, as the state space heuristics can be translated to action set heuristics, and action set heuristics can be directly used as state space heuristics. In other words, we learn a more general form of heuristics than existing literature.

The earliest work on learning heuristics through neural networks that we are aware of is by Ernandes and Gori (2004). They learned heuristics to solve the n-puzzle problem, where the goal is to move a set of tiles in a grid to reach a goal configuration. They used a neural network to learn a heuristic by encoding the state of the grid as a feature vector. Later on, Yoon, Fern, and Givan (2008) extracted features from the relaxed plan obtained when computing the h^{FF} heuristic and used it to train a heuristic. Samadi, Felner, and Schaeffer (2008) learned heuristics whose input features are values of multiple existing heuristics, and use a neural network to combine these heuristic values together. They also modified the loss function to penalise heuristic over-estimations more, thereby making the resulting heuristic closer to an admissible one.

Beyond the initial interest in learning heuristics for planning, some recent works have focused on learning domain independent heuristics. In our case, our learned action set heuristics are trained in a domain-independent way, but the resulting heuristics are domain specific. Works on learning domain independent heuristics focus on learning a single heuristic that can be used in any domain, in particular ones not seen during training. The first work to do so by Gomoluch et al. (2017) uses handcrafted features from planning tasks and values of the h^{FF} and h^{eca} heuristic (Helmert and Geffner, 2008). However, their evaluation is only on a small number tasks and their performance gain over the h^{FF} heuristic is marginal. STRIPS-HGN (Shen, Trevizan, and Thiébaux, 2020) is the first work to learn domain-independent heuristics from scratch and showed promising results where it is more informed than $h^{\max}$ on certain domains. More recently, (Chen, Thiébaux, and Trevizan, 2024) learned domain-independent heuristics using graph neural networks and showed that their learned heuristics are more much more informed than STRIPS-HGN, although still being significantly weaker than the h^{FF} heuristic. To our knowledge, this is the state-of-the-art in learning domain-independent heuristics. Our work is different from these works in that we learn domain-specific heuristics in a domain-independent way. Unlike these works, our heuristics need to be trained for each domain, and cannot be used in unseen domains. However, our learned heuristics are much more informed than the h^{FF} heuristic, and consequently significantly ahead of the state-of-the-art in learning domain-independent heuristics.

Returning to learning domain-specific heuristics, bootstrapping is also a common idea studied in a number of works. Roughly, bootstrapping means to start with a weak heuristic function and iteratively improve it. The first work to do so is Arfaee, Zilles, and Holte (2011), where they start with a weak heuristic h_0 and used it to providing training samples to learn a new heuristic h_1. They then repeat this process, iterating between training sample generation and learning from new samples until obtaining a sufficiently strong heuristic. More recently, Ferber et al. (2022) performed bootstrapping with neural networks in three ways: (1) a similar method to Arfaee, Zilles, and Holte (2011) but with different state representations and deeper neural networks, (2) bootstrapping to estimate search space size, and (3) bootstrapping with approximate value iteration, a common technique in reinforcement learning. A major limitation of these works is that their learned heuristic functions are not only domain-specific, but also instance-specific.

In other words, their learned heuristics are not generalisable to new instances of the same domain. Leapfrogging is an extension of bootstrapping to allow for generalisation to new instances with a domain. Karia and Srivastava (2021) used leapfrogging to train neural networks that predict, at the same time, a heuristic value and a ground action to apply. These ideas are all interesting and represent how a learned heuristic can be iteratively improved. However, their performance all lag behind state-of-the-art classical planners like LAMA-first. Our contributions are empirically much stronger than these works, and our methods are not necessarily in conflict with these ideas. It is interesting to consider how bootstrapping and leapfrogging can be combined with our learned action set heuristics to further improve planning performance.

A key reason why leapfrogging and bootstrapping are theoretically interesting is that they allow generation of training samples in an unsupervised way from an existing heuristic function. Like we had discussed in Section 2.3.4, training heuristics through ranking also allows the use of much more training samples. We will not repeat the discussion on these ideas here, but it is important to note that prior works ((Garrett, Kaelbling, and Lozano-Pérez, 2016; Chrestien et al., 2023; Hao et al., 2024; Chen and Thiébaux, 2024)) have all discussed learning heuristics through ranking and achieved different levels of empirical success. Our works builds upon their ideas. Specifically, our way of training heuristics builds upon the latest work by Chen and Thiébaux (2024), and extends it to train action set heuristics and use importance values for our training samples. Implementation-wise, our Rank2Plan library (introduced in Section 5.1.5), makes it computationally easier to train heuristics through ranking.

Lastly, we want to note that there is a wide variety of works that we have not covered yet, from which we would like to highlight a few. Francès et al. (2019) learned, for a few simple domains, potential heuristics (Pommerening et al., 2015) that they manually proved to be descending and dead-end avoiding (Seipp et al., 2016). These heuristics are therefore guaranteed to find a plan for their domains in polynomial time with greedy search. Outside classical planning, Chen and Thiébaux (2024) extended the WL method (Chen, Trevizan, and Thiébaux, 2024) to account for continuous state variables and learned very informed heuristics for numeric planning. These works represent interesting directions of research for learning heuristics, namely obtaining guarantees on the heuristic and extending learning to more expressive forms of planning beyond classical planning.

6.2 Learning Generalised Policies for Planning

A policy for a planning task is a function π that maps states in $\mathbb{S}$ to ground actions in $\mathbb{A}$. Typically, policies are used to directly traverse the state space of a planning task by simply always applying the action that the policy suggests. The hope is that by doing this, we eventually reach a goal state. A generalised policy is a policy that is defined for states on all tasks under a certain domain. A number of works have studied learning generalised policies for planning. Before discussing these works, we note that any state space heuristic can also be used to define policies, where one simply always return the

action that leads to the successor state with the lowest heuristic value.

The most pioneering work in learning generalised policies for planning is ASNets (Toyer et al., 2020). ASNets is a graph neural network where the graph consists of ground atoms and actions. It is able to generalise within a domain by having update functions in the GNN that only depend on action schemas and predicates. ASNets were shown to be able to learn policies that can solve arbitrarily large tasks in a domain. ASNets have also been extended to numeric planning, yielding state-of-the-art results (Wang and Thiébaux, 2024). Around the same time, Groshev et al. (2018) learned policies for Sokoban and the travelling salesperson problem (TSP) using graph neural networks.

More recently, Ståhlberg, Bonet, and Geffner (2022a) learned optimal general policies using graph neural networks, subject to constraints on the domain. Later, Ståhlberg, Bonet, and Geffner (2022c) used a similar approach to learn general policies that are not optimal, but in an unsupervised way.

6.3 Other Methods to Aid Planning with Learning

Various interesting ideas have also been developed to aid planning with learning. These methods are related to our work in that they are in the reverse direction of our contribution of partial space search, which aids learning by adapting planning to make the best use of learning methods.

A simple idea for this is to learn to predict an ideal planner for solving task, since usually not a single planner is ideal for all tasks. Katz et al. (2018) is one such system that learns to predict the ideal planner for a task. They do so by training a convolutional neural network on the adjacency matrix representation of a graph representation of the planning task. Later, (Ma et al., 2020) built on their work by using graph neural networks instead and also introduced adaptive scheduling, where the chosen planner is changed mid-task if it exceeds a time limit. All these methods rely on the underlying planner, and are as performant as the planners they are applied to.

(Gnad et al., 2019) also learned to partially ground planning tasks. They use various classical machine learning techniques to predict parts of the problem that need to be grounded. This way, they reduce the search space and grounding time of the planner. Since their method does not guarantee grounding all parts of the problem that are necessary, they incrementally ground larger parts of the problem if the planner fails to find a plan. As shown in the International Planning Competition 2023 (Taitler et al., 2024), their method actually ends up reducing the overall performance of the planner.

PLOI (Silver et al., 2021) takes an alternative approach of learning to predict which objects are irrelevant to solving the planning task at hand. As seen in our blocksoworld-large domain, not all objects are necessary to solve a task, and PLOI uses a GNN to score the usefulness of each object. They incrementally plan with larger sets of objects until a plan is found. Their method works with large number of objects but few of

them relevant to the goal. This achieves similar results to partial space search on the blocksoworld-large domain. However, their method has the drawbacks that PLOI is specialised to dealing with irrelevant objects and causes incompleteness in the search space. Partial space search, although not necessarily as effective as PLOI on mitigating the impact of irrelevant objects, is more general and preserves completeness.

6.4 Planning Model Design with Learning

A key challenge in applying planning in practice is the difficulty and error-prone nature of designing planning models. Model design is usually done by hand, and it can be easy to make errors that cause the domain to not model the real world as intended. It is also easy to not account for unseen obstacles in the real world. The ultimate result that the obtained plans from these incorrect models cannot be actually applied in the real world with the intended results. Learning methods can be used to aid in safe planning model design by automatically learning the model from data.

Examples of works in this direction include Stern and Juba (2017) and Juba, Le, and Stern (2021), where they learned action models from example plans. Their work has also been extended to work with stochastic world models (Juba and Stern, 2022) and numeric world models (Mordoch, Stern, and Juba, 2023).

It is also possible to learn world models from other sources of data. (Bonet and Geffner, 2020) learned world models from the structure of the state space. They do so by using a SAT theory encoding of the world model controlled by a number of hyperparameters. They then learned the hyperparameters that satisfy the structure of the state space while leading to the simplest planning model. More recently, Xi, Gould, and Thiébaux (2024) learned planning models through visual traces of plans in a neuro-symbolic approach.

6.5 Search Reduction Techniques

Search reduction techniques are methods that aim to reduce the search space of planning tasks, or more broadly, reduce the amount of computation needed to search for a solution. These methods are related to our work in that partial space search fits the broad definition of search reduction techniques — it does not reduce the search space, but factorises it in a way that allows for more efficient search.

A common search reduction technique is the use of deferred heuristic evaluation with preferred operators. This is in particular the main search reduction technique used in the state-of-the-art LAMA planner (Richter and Westphal, 2010). We had introduced deferred evaluation (lazy search) in Section 2.2.1, where the heuristic value of a search node is not computed until the node is expanded, and instead the priority value of search nodes in the search queue is the heuristic value of the parent node. This reduces the number of heuristic evaluations needed to find a plan when there are more nodes generated than expanded. Preferred operators are ground actions (operators) that are

deemed particularly useful in a state. In LAMA, these are computed as side products of computing heuristic values. States that are reached by applying preferred operators are expanded earlier. Preferred operators are very synergistic with deferred evaluation. States reached by preferred operators are generally better, and expanding them earlier means we often never have to evaluate many states that are not reached by preferred operators, since they are only generated but not expanded. In practice, this is very effective, as shown by the success of LAMA. However, partial space search is a more general method for dealing with branching factor that does not require dedicated ways for computing preferred operators. Moreover, it is entirely feasible to apply deferred evaluation with preferred operators with partial space search, given ways to generate preferred transitions in partial space search.

There have also been a wealth of search reduction techniques that actually reduce the size of the search space. In many planning tasks, many actions can be interleaved independently, resulting in a blowup in the branching factor of the state space. Partial order reduction techniques exploit this property to select only a subset of the applicable actions in a state while preserving completeness of the search process. We refer to the papers Chen and Yao (2009) and Wehrle and Helmert (2012, 2014) for more detail on these techniques. Symmetry reduction techniques exploit the symmetric nature of planning tasks, identifying symmetric actions or states and pruning them away. We refer to the papers Fox and Long (1999); Pochter, Zohar, and Rosenschein (2011), and Domshlak, Katz, and Shleyfman (2012) for more detail on these techniques. Lastly, planning problems have also been translated to problems called the Petri net reachability (unfolding) problems, which recognises the independent subproblems and allows for their separate resolution Hickmott et al. (2007).

Research interest in Petri net for planning has died out in recent years. However, a similar technique called star-topology decoupled search, or simply decoupled search, has been developed in recent years and gathered significant research interest (Gnad and Hoffmann, 2018). Breaks planning tasks into components, with a central component and multiple leaf components. Leaf components can only interact via the central component. Decoupled search thereby factorises the planning task such that one only has to consider the dependency of leaf components on the centre component, but not between leaf components. This ultimately allows for reduction in the search space of planning tasks (Speck and Gnad, 2024). Decoupled search has been shown to be related to Petri net unfolding, and dominate each other under different conditions (Gnad and Hoffmann, 2019).

Our work of partial space search is generally compatible with the above search reduction techniques. Partial space search is ultimately a tree-factoring on the action space. In general, partial space search can be applied after performing search space reduction, to further factorise the resulting actions.

Lastly, we want to discuss a specific search reduction technique proposed for solving constraint satisfaction problems, due to its similarity to our work. Botea and Bulitko

(2022) introduces tiered state expansion. There, they focus on solving constraint satisfaction problems where states are defined by a finite set of variables with a finite discrete domain each. Their states are instantiations of zero or more variables, and successors of states are obtained by instantiating one more variable. Moreover, legal states need to satisfy a number of constraints on state variables. Whenever expanding a state, they classify the successors into two tiers: tier-1 successors are those that are preferred (i.e., have high reward), and tier-2 successors are those that are not preferred. They then only immediately add tier-1 successors and a special placeholder successor to the search queue. When the placeholder successor is expanded, it is replaced by the tier-2 successors. This way, they reduce the branching factor of the search space. Moreover, by the time tier-2 successors are expanded, constraint propagation can hopefully prune away many of them.

Their method is similar to partial space search in that they group actions together, while partial space search factorises actions into action sets (partial actions) in a tree structure. Their method is akin to a merge of partial space search and deferred evaluation with preferred operators, applied to constraint satisfaction problems. However, their work only discusses how to classify successors into tiers for a specific problem, the Romanian Crossword. Our work is much more general in that it can be applied to any classical planning task.

Nonetheless, their idea of grouping actions into tiers and expanding lower tiers later is interesting. It is possible to imagine this idea being applied to planning, where action set heuristics are used to give heuristic values to tiered sets of actions. This is an interesting direction for future work.

Chapter 7

Conclusion

This chapter summarises the work and contributions of this thesis, discusses its limitations, and suggest a wide number of directions for future work.

7.1 Contributions

The goal of this work is to build a planning system that not only uses learning, but where the learning and planning processes are integrated in a way that they work better together. We have made a number of contributions towards this goal from both the learning and planning perspectives.

From the planning perspective, we have developed the new search space, partial space search, that allows for a more focused and efficient search process. Partial space search is able to reduce the branching factor of the search space significantly and is particularly well-suited for high branching factor tasks. In doing so, partial space search is designed for the performance profiles of learned heuristics, which are more accurate but potentially slower. Moreover, partial space search allows the generation of larger training datasets for learning heuristics, which is fundamental for the success of learning heuristics. In order to guide partial space search, we formalised the notion of action set heuristics, which are heuristics that estimate the quality of a set of actions in a state, rather than just the state itself.

From the learning perspective, we have introduced how to learn action set heuristics using graph representations. We showed how to leverage partial space search to generate large training datasets and how to use these training datasets to efficiently learn powerful action set heuristics. Experiments showed that these learned action set heuristics are the strongest learned heuristics to date, even if not used with partial space search.

We integrated our above contributions into a new planning system, LazyLifted, that

uses learned action set heuristics and partial space search together. We showed that empirically LazyLifted outperforms the state-of-the-art planning systems under various categories, specifically GOOSE for learning-for-planning, Powerlifted for lifted planning, and most importantly, LAMA-first for classical planning in general. Our strong empirical results validate that integrating learning and planning can lead to more effective planning.

We also have additional contributions. In terms of action set heuristics, we showed how to automatically translate any existing state space heuristic to an action set heuristic and how to do so for the FF heuristic efficiently. In terms of software engineering, we have developed the performant and maintainable LazyLifted planning system, which will be open-source and available for use by the community. In terms of training, we have developed a new Python library, Rank2Plan, for training planning heuristics through ranking. And lastly, in terms of benchmarks, we have developed a number of benchmarks for examining learning for planning systems on high branching factor tasks.

7.2 Limitations

Our work is not perfect. The most important limitation of our work is the requirement by partial space search for specialised action set heuristics. This makes it harder to integrate partial space search into existing planning systems and employ existing techniques with it, since most such techniques are designed for state space search. Although we have developed a method to automatically translate state space heuristics to action set heuristics, state-of-the-art planning approaches employ more sophisticated techniques. We have not explored how these techniques can be adapted to work with partial space search. We have also not explored how prominent state space heuristics can be adapted to work with partial space search in a more organic way that is more computationally efficient and informed.

Another limitation of our work is the relatively poor performance of our systems on certain tasks such as rovers and satellite. We believe that this is due to the limited expressivity of our features for training action set heuristics. We can only run the WL algorithm for a fixed number of iterations, and due to computational reasons this number is typically small. This limits the complexity of features we obtain from the WL algorithm, and hence the performance of our learned heuristics. Nonetheless, planning methods usually have different strength and weaknesses, and it is hard to develop a single planning system that is strong on all tasks.

It is important to also address the domain specific nature of our training method. We obtain heuristics by training on a set of training problems on a fixed domain, and the obtained heuristic can only be applicable within the domain. It is worth noting however, that we had discussed works such as Chen, Thiébaux, and Trevizan (2024), that have learned domain-independent heuristics. In principle, their methods are compatible with our methods.

7.3 Future Work

Our work marks a major step in the field of integrating learning and planning. We have made the first step in exploring how planning and learning can work better together for more effective planning. However, there are many opportunities for further research, both continuing the work we have started and exploring new directions. For the rest of this section, we will discuss these opportunities for future work in order of increasing complexity and importance to the field of planning.

7.3.1 Additional engineering

Like we had discussed, our implementation of the LazyLifted planning has been in general well-engineered and even outperforms the state-of-the-art Powerlifted planning in terms of search speed. However, there are still additional ways to make the implementation more efficient and accessible.

Specifically, we had discussed in Section 5.1.1 that our implementation is still not as memory efficient as similar planners. Additional work can be put in to make the implementation more memory efficient, in particular by investigating what existing techniques can be applied to our implementation.

Additionally, we had discussed in Section 5.1.2 that maintainability and ease of use are important design goals for our implementation. We can further improve the maintainability of our implementation by adding more documentation and tests, and by making the code more modular and extensible. Moreover, given our implementation is already reasonably efficient, we can also focus on turning our implementation into a library form that can be more easily integrated into other projects. This would in particular mean that future research work can more easily build on top of our implementation by using the library, rather than having to implement on the basis of our code.

7.3.2 Additional techniques surrounding partial space search

We have developed partial space search and made the first step in adapting existing planning techniques, namely heuristics, to work with it. However, there are many more planning techniques that can be adapted to work with partial space search. For example, we have not explored how to adapt preferred operators like techniques to work with partial space search. Future work can explore how to adapt these techniques to work with partial space search, and how to design new native techniques, such as dedicated and non-learned action set heuristics, that work well with partial space search.

7.3.3 Theoretical analysis

Previous works in learning for planning, such as Chen, Trevizan, and Thiébaux (2024) and Ståhlberg, Bonet, and Geffner (2022a), have devoted considerable attention on the theoretical nature of their contributions. Due to time constraints, we have not explored

deeply the theoretical properties of our work. In particular, we only showed the basic correctness of partial space search, but did not have theoretical results surrounding the performance of partial space search. We also did not explore the expressivity limitations of our graph representations at all like prior works (e.g., Chen, Trevizan, and Thiébaux (2024)) had. Future work can explore these aspects. We expect this to be a minor work, in particular as there already exist results on the expressivity of graph based learning (Morris et al., 2019; Barceló et al., 2020) and these results have been applied in planning (Chen, Trevizan, and Thiébaux, 2024).

A more interesting work than exploring the theoretical limits of our work, however, is to explore how we can adapt our methods to achieve high theoretical limits, which should hopefully translate to better performance in practice. This may mean modifying our graph representations to be more expressive, or modifying the process by which we learn heuristics from graph representations.

7.3.4 Continuous learning

Our methods, and the existing state-of-the-art methods for learning planning heuristics, all train heuristics in a single-step manner. That is, they train heuristics once and then use them for the rest of the planning process. However, it is possible that the heuristics can be improved further if they are trained iteratively, or that they are able to gradually learn and improve as they are applied to more and more problems.

We had discussed in our related works (Section 6.1) methods to iteratively improve learned heuristics through techniques such as bootstrapping and leapfrogging. It is interesting future work explore how they can be best applied to our methods, and how effective they are in practice.

Learning while planning is perhaps a more open question than iterative learning. Ideally, learned heuristics should be able to take of advantage of experiences from all past planning problems that it was either trained on or used on. To our knowledge, no existing work in planning has explored how to do this. Future work can potentially start simply adding plans generated by the current learned heuristic to the set of training plans, and then retraining on the larger set of training plans. This is a simple starting point that is likely inefficient in practice.

In general, we believe continuous learning to be a promising direction for future work with a lot of potential work to be done. Successful continuous learning methods could allow learning for planning systems to be much more scalable in practice.

7.3.5 Efficient pattern extraction

In effect, what the WL algorithm does when learning planning heuristics is to extract a number of patterns on a graph, which are used as features for the heuristic. The problem with current methods is that the result weight matrix is very sparse, meaning most patterns are not actually useful for the heuristic. In future work, it is interesting

to explore how we can extract patterns more efficiently, such that the result set of patterns are more useful for the heuristic. This could potentially involve adapting the WL algorithm or using more sophisticated graph neural network methods. More generally, this ties into the active research area of graph learning, with the specific application to planning.

A key reason why efficient pattern extraction is important is that our current pattern extraction method (WL) limits the expressivity of the patterns we can use. Specifically, our experiments used only 2 iterations of the WL algorithm, so as to make the number of colours tractable, where colours in WL correspond to patterns in general. This is despite the fact that most colours are not actually useful for the heuristic. If we can extract patterns more efficiently, we can afford to use much more expressive and complex patterns for our heuristics. We expect this to lead to fundamental performance improvements in our methods.

7.3.6 Learning search space reductions

Partial space search fits under the broad umbrella of search space reduction techniques. Search space reduction is interesting and important to planning as it represents a fundamental reduction in the effort needed by search algorithms, and is orthogonal to the research for better heuristics. There have been many works in the past that have explored search reduction techniques for planning, such as symmetry reduction, partial order reduction, and decoupled search. We had discussed these in Section 6.5. There have also been works on learning search reduction techniques, which we discussed in Section 6.3, but these works have not shown to be very beneficial empirically.

We believe that learning search reductions is a promising direction for future work. An easy example of a search reduction that can be learned is symmetry. The WL algorithm was originally designed for identify isomorphic graphs. Given that we represent search states and nodes as graphs, it is straightforward how we can use the WL algorithm to identify symmetries in the search space. This naive approach would render the search process incomplete, in that the WL algorithm may identify non-symmetric search nodes as symmetric. More sophisticated approaches may focus on learning symmetries that are guaranteed to be correct, or more accurate symmetries that are not necessarily guaranteed to be correct.

A more sophisticated example of future work that learns search reduction is to learn a model that identifies subproblems of a planning problem and what order these subproblems should be approached in. This may allow planners to always focus on a particular subproblem at a time, rather than the whole complex planning task. This is similar to the idea of hierarchical planning, where a planner first solves a high-level planning task and then refines the resulting high-level plan into a low-level plan. Here, we suggest learning how to identify the high-level planning task automatically from the low-level planning task.

Another interesting direction for learning search space reduction is to extend the idea

of tiered state expansions that we had discussed in Section 6.5. The idea is to learn a model that can group the applicable actions of a state into tiers, where top-tier actions are treated as usual, and actions in lower tieres are represented as placeholder node that is inserted into the search queue. Once expanded, this placeholder node expands into the successors of applying the actions in the lower tier to the original state. Future work may investigate how to learn such a model for grouping actions. The evaluation of the placeholder node for heuristic search may be done by action set heuristics that we had developed.

Lastly, we want to discuss the prospect of learning macro operators. Macro operators represent the merger of a number of action schemas into a single action schema, representing the sequential application of the actions in the macro operator. The motivation is that macro operators represent abstractions that reduce the computational effort for repetitive applications of several actions together. Learning macro operators were historically researched in works such as Botea et al. (2005), and research interests in them have died out in the recent decade, partly due to the lack of empirical success. We believe that using advances in learning, future works can potentially learn better macro operators that are more effective in practice. Moreover, we believe macro operators compliment partial space search nicely, in that the macro operators typically have high branching factors, that can be reduced by partial space search. This way, we are able to reap the benefit of macro operators without paying its price.

Our work is on deeply integrating learning and planning. In general, all of these directions represent deeper integrations of learning and planning, and we believe they are promising directions for future work.

7.3.7 Learning for more expressive planning

Classical planning has restricted expressivity, and many real-world planning problems require reasoning about concepts such as time, continuous values, and uncertainty. There have considerable research in these more expressive forms of planning. We refer to Geffner and Bonet (2013) for a comprehensive review of these works. However, there has been less research on how learning can be applied to these more expressive forms of planning. This is despite the fact that learning methods are often more flexible than non-learning methods, and are hence likely easier to extend to more expressive forms of planning. This is demonstrated in works by Wang and Thiébaux (2024) and Chen and Thiébaux (2024), where they extended learning methods for classical planning to numeric planning (Fox and Long, 2003) in straightforward ways. Future work can focus on how to extend learning methods to more expressive forms of planning in more sophisticated ways. In particular, we believe our work in this thesis extends trivially to numeric planning via methods proposed by Chen and Thiébaux (2024).

Beyond numeric planning, it is also interesting to explore how learning, and our method, can be applied to temporal planning and various forms of planning with uncertainties. This is in particular, very applicable for tasks with natural relational action structures.

There, partial space search may be directly applicable and may help provide a more efficient search process. Moreover, the concept of action set heuristics may also be useful. For example, it is possible to imagine a form of planning where actions are parameterised by continuous values. The concept of action set heuristics can be extended to this form of planning to reduce the branching factor of the search space from infinite to finite.

7.4 Final Remarks

Learning and planning are two fundamental aspects of artificial intelligence. The rise of deep learning has led to significant interest in using learning for planning. Our work in this thesis is a step towards adapting planning systems to take the best advantage of learning. We have developed a new planning system, LazyLifted, that uses learning and planning systems designed for each other. We showed that LazyLifted outperforms the state-of-the-art planning systems in various benchmarks. This highlights the potential of deeply integrating learning and planning, with some promising directions for future research being discussed in our future works section.

Bibliography

2008. *Proc. of 18th Int. Conf. on Automated Planning and Scheduling (ICAPS)*. [Cited on pages 107 and 108.]

2008. *Proc. of 23rd AAAI Conference on Artificial Intelligence*. [Cited on pages 107 and 109.]

2009. *Proc. of 21st Int. Joint Conf. on AI (IJCAI)*. [Cited on pages 104 and 108.]

2012. *Proc. of 22nd Int. Conf. on Automated Planning and Scheduling (ICAPS)*. [Cited on pages 105, 107, and 111.]

2016. *Proc. of 25th Int. Joint Conf. on AI (IJCAI)*. [Cited on pages 106 and 110.]

2022. *Proc. of 32nd Int. Conf. on Automated Planning and Scheduling (ICAPS)*. [Cited on pages 105 and 110.]

2024. *Proc. of 34th Int. Conf. on Automated Planning and Scheduling (ICAPS)*. [Cited on pages 104 and 110.]

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley. [Cited on page 17.]

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning Heuristic Functions for Large State Spaces. *Artif. Intell.*, 175: 2075–2098. [Cited on page 88.]

Barceló, P.; Kostylev, E.; Monet, M.; Pérez, J.; Reutter, J.; and Silva, J.-P. 2020. The Logical Expressiveness of Graph Neural Networks. In *Proc. of 8th. Int. Conf. on Learning Representations (ICLR)*. [Cited on pages 20 and 98.]

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *Artif. Intell.*, 129(1): 5–33. [Cited on page 14.]

Bonet, B.; and Geffner, H. 2020. Learning First-Order Symbolic Representations for Planning from the Structure of the State Space. In *Proc. of 24th European Conference on Artificial Intelligence (ECAI)*, 2322–2329. [Cited on page 91.]

Botea, A.; and Bulitko, V. 2022. Tiered State Expansion in Optimisation Crosswords. In *Proc. of 18th AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. [Cited on page 92.]

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *J. Artif. Intell. Res.*, 24: 581–621. [Cited on page 100.]

Cai, J.-Y.; Fürer, M.; and Immerman, N. 1992. An Optimal Lower Bound on the Number of Variables for Graph Identification. *Combinatorica*, 12(4): 389–410. [Cited on page 21.]

Chen, D. 2023. *GOOSE: Learning Heuristics and Parallelising Search for Grounded and Lifted Planning*. Honours thesis, Australian National University. [Cited on pages xii and 60.]

Chen, D. Z.; and Thiébaux, S. 2024. Learning Heuristics for Numeric Planning. To Appear in 38th Int. Conf. on Neural Information Processing Systems (NeurIPS 2024). [Cited on pages 19, 23, 24, 25, 41, 89, and 100.]

Chen, D. Z.; Thiébaux, S.; and Trevizan, F. 2024. Learning Domain-Independent Heuristics for Grounded and Lifted Planning. In *Proc. of 38th AAAI Conference on Artificial Intelligence*, 20078–20086. [Cited on pages 2, 88, and 96.]

Chen, D. Z.; Trevizan, F.; and Thiébaux, S. 2024. Return to Tradition: Learning Reliable Heuristics with Classical Machine Learning. In ica (2024), 68–76. [Cited on pages 2, 21, 22, 23, 24, 41, 47, 51, 65, 89, 97, and 98.]

Chen, Y.; and Yao, G. 2009. Completeness and Optimality Preserving Reduction for Planning. In ijc (2009), 1659–1664. [Cited on page 92.]

Chrestien, L.; Pevný, T.; Edelkamp, S.; and Komenda, A. 2023. Optimize Planning Heuristics to Rank, not to Estimate Cost-to-Goal. In *Proc. of 37th Int. Conf. on Neural Information Processing Systems (NeurIPS)*. [Cited on pages 19, 24, and 89.]

Corrêa, A. B.; Francès, G.; Pommerening, F.; and Helmert, M. 2021. Delete-Relaxation Heuristics for Lifted Classical Planning. In *Proc. of 31st Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 94–102. [Cited on pages 17 and 18.]

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In *Proc. of 30nd Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 80–89. [Cited on pages 3, 13, and 52.]

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2022. The FF Heuristic for Lifted Classical Planning. In *Proc. of 36th AAAI Conference on Artificial Intelligence*, 9716–9723. [Cited on pages 17, 18, and 40.]

Corrêa, A. B.; and Seipp, J. 2022. Best-First Width Search for Lifted Classical Planning. In *Proc. of 32nd Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 11–15. [Cited on page 13.]

Culberson, J. C. 1997. Sokoban is PSPACE-complete. Technical Report TR 97-02, Department of Computing Science, The University of Alberta, Edmonton, Alberta, Canada. [Cited on page 60.]

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3): 374–425. [Cited on page 17.]

Dedieu, A.; Mazumder, R.; and Wang, H. 2022. Solving L1-regularized SVMs and Related Linear Programs: Revisiting the Effectiveness of Column and Constraint Generation. *J. Mach. Learn. Res.*, 23: 1–41. [Cited on pages 4 and 56.]

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced Symmetry Breaking in Cost-Optimal Planning as Forward Search. In ica (2012), 343–347. [Cited on page 92.]

Drexler, D.; Gnad, D.; Höft, P.; Seipp, J.; Speck, D.; and Ståhlberg, S. 2023. Ragnarok. In *IPC-10 Planner Abstracts*. [Cited on page 10.]

Drexler, D.; Seipp, J.; and Geffner, H. 2023. Learning Hierarchical Policies by Iteratively Reducing the Width of Sketch Rules. In *Proc. of 20th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, 208–218. [Cited on page 2.]

Edelkamp, S. 2001. Planning with Pattern Databases. In *Proc. of 6th European Conference on Planning (ECP)*, 84–90. [Cited on page 16.]

Ernandes, M.; and Gori, M. 2004. Likely-admissible and Sub-symbolic Heuristics. In *Proc. of the 16th European Conf. on Artificial Intelligence (ECAI)*, 613–617. [Cited on page 88.]

Fan, W.; Ma, Y.; Li, Q.; He, Y.; Zhao, E.; Tang, J.; and Yin, D. 2019. Graph Neural Networks for Social Recommendation. In *World Wide Web Conf.*, 417–426. [Cited on page 20.]

Ferber, P.; Geißer, F.; Trevizan, F.; Helmert, M.; and Hoffmann, J. 2022. Neural Network Heuristic Functions for Classical Planning: Bootstrapping and Comparison to Other Methods. In ica (2022), 583–587. [Cited on pages 19, 24, and 88.]

Fox, M.; and Long, D. 1999. The Detection and Exploitation of Symmetry in Planning Problems. In *Proc. of 16th Int. Joint Conf. on AI (IJCAI)*, 956–961. [Cited on page 92.]

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res.*, 20: 61–124. [Cited on page 100.]

Francès, G.; Corrêa, A. B.; Geissmann, C.; and Pommerening, F. 2019. Generalized Potential Heuristics for Classical Planning. In *Proc. of 28th Int. Joint Conf. on AI (IJCAI)*, 5554–5561. [Cited on page 89.]

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. [Cited on page 54.]

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2016. Learning to Rank for Synthesizing Planning Heuristics. In ijc (2016), 3089–3095. [Cited on pages 19, 24, and 89.]

Geffner, H. 2018. Model-free, Model-based, and General Intelligence. In *Proc. of 27th Int. Joint Conf. on AI (IJCAI)*. [Cited on page 1.]

Geffner, H.; and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*, volume 7 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool. [Cited on pages 1, 7, 8, and 100.]

Gnad, D.; and Hoffmann, J. 2018. Star-Topology Decoupled State Space Search. *Artif. Intell.*, 257: 24–60. [Cited on page 92.]

Gnad, D.; and Hoffmann, J. 2019. On the Relation between Star-Topology Decoupling and Petri Net Unfolding. In *Proc. of 29th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 172–180. [Cited on page 92.]

Gnad, D.; Torralba, Á.; Domínguez, M. A.; Areces, C.; and Bustos, F. 2019. Learning How to Ground a Plan – Partial Grounding in Classical Planning. In *Proc. of 33rd AAAI Conference on Artificial Intelligence*, 7602–7609. [Cited on pages 2 and 90.]

Gomoluch, P.; Alrajeh, D.; Russo, A.; and Bucchiarone, A. 2017. Towards Learning Domain-Independent Planning Heuristics. In *IJCAI 2017 Workshop on Architectures for Generality and Autonomy*. [Cited on page 88.]

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *Proc. of 28th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 408–416. [Cited on page 90.]

Gupta, N.; and Nau, D. S. 1992. On the Complexity of Blocks-World Planning. *Artif. Intell.*, 56(2–3): 223–254. [Cited on page 58.]

Hamilton, W. 2020. *Graph Representation Learning*, volume 14 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool. [Cited on page 20.]

Hao, M.; Trevizan, F.; Thiébaux, S.; Ferber, P.; and Hoffmann, J. 2024. Guiding GBFS through Learned Pairwise Rankings. In *Proc. of 33rd Int. Joint Conf. on AI (IJCAI)*. [Cited on pages 2, 19, 24, 49, and 89.]

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proc. of 22nd AAAI Conference on Artificial Intelligence*, 1007–1012. [Cited on page 16.]

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*, volume 13 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool. [Cited on pages 8 and 27.]

Haslum, P.; Slaney, J.; and Thiébaux, S. 2012. Minimal Landmarks for Optimal Delete-Free Planning. In ica (2012), 353–357. [Cited on page 14.]

Hearn, R. A.; and Demaine, E. D. 2005. PSPACE-Completeness of Sliding-Block Puzzles and Other Problems through the Nondeterministic Constraint Logic Model of Computation. *Theoretical Computer Science*, 343(1–2): 72–96. [Cited on page 60.]

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26: 191–246. [Cited on pages 1, 13, and 51.]

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artif. Intell.*, 173: 503–535. [Cited on page 18.]

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. of 19th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 162–169. [Cited on page 17.]

Helmert, M.; and Geffner, H. 2008. Unifying the Causal Graph and Additive Heuristics. In ica (2008), 140–147. [Cited on page 88.]

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *Proc. of 17th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 176–183. [Cited on page 16.]

Helmert, M.; and Röger, G. 2008. How Good is Almost Perfect? In aaa (2008), 944–949. [Cited on page 19.]

Hickmott, S. L.; Rintanen, J.; Thiébaux, S.; and White, L. B. 2007. Planning via Petri Net Unfolding. In *Proc. of 20th Int. Joint Conf. on AI (IJCAI)*, 1904–1911. [Cited on page 92.]

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *J. Artif. Intell. Res.*, 14: 253–302. [Cited on page 15.]

Joachims, T. 2002. Optimizing Search Engines using Clickthrough Data. In *Proc. of 8th ACM Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*. [Cited on page 49.]

Juba, B.; Le, H. S.; and Stern, R. 2021. Safe Learning of Lifted Action Models. In *Proc. of 18th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*. [Cited on page 91.]

Juba, B.; and Stern, R. 2022. Learning Probably Approximately Coomplete and Safe Action Models for Stochastic Worlds. In *Proc. of 36th AAAI Conf. on Artificial Intelligence (AAAI)*. [Cited on page 91.]

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *Proc. of 35th AAAI Conference on Artificial Intelligence*, 8064–8073. [Cited on page 89.]

Karpas, E.; and Domshlak, C. 2009. Cost-Optimal Planning with Landmarks. In ijc (2009), 1728–1733. [Cited on page 17.]

Katz, M.; and Domshlak, C. 2008. Optimal Additive Composition of Abstraction-based Admissible Heuristics. In ica (2008), 174–181. [Cited on page 16.]

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online Planner Selection for Cost-Optimal Planning. In *IPC-9 Planner Abstracts*, 57–64. [Cited on page 90.]

Keyder, E.; and Geffner, H. 2008. Heuristics for Planning with Action Costs Revisited. In *Proc. of 18th European Conference on Artificial Intelligence (ECAI)*, 588–592. [Cited on pages 15 and 16.]

Kiefer, S.; and McKay, B. D. 2020. The Iteration Number of Colour Refinement. In *Proc. 47th Int. Colloquium on Automata, Languages, and Programming (ICALP)*. [Cited on page 21.]

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening A*. *Artif. Intell.*, 129: 199–218. [Cited on page 12.]

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25 (NIPS 2012)*. [Cited on page 1.]

Kurniawati, H.; Hsu, D.; and Lee, W. S. 2008. SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces. In *Robotics: Science and Systems IV*. [Cited on page 1.]

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In *Proc. of 20th European Conference on Artificial Intelligence (ECAI)*, 540–545. [Cited on page 13.]

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Online Planner Selection with Graph Neural Networks and Adaptive Scheduling. In *Proc. of 34th AAAI Conference on Artificial Intelligence*, 5077–5084. [Cited on page 90.]

Mordoch, A.; Stern, R.; and Juba, B. 2023. Learning Safe Numeric Action Models. In *Proc. of 37th AAAI Conf. on Artificial Intelligence (AAAI)*. [Cited on page 91.]

Morris, C.; Ritzert, M.; Fey, M.; Hamilton, W. L.; Lenssen, J. E.; Rattan, G.; and Grohe, M. 2019. Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks. In *Proc. of 33rd AAAI Conf. on Artificial Intelligence (AAAI)*. [Cited on pages 20 and 98.]

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley. [Cited on page 11.]

Penberthy, J. S.; and Weld, D. S. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proc. of 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, 103–114. [Cited on page 10.]

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting Problem Symmetries in State-Based Planners. In *Proc. of 25th AAAI Conference on Artificial Intelligence*, 1004–1009. [Cited on page 92.]

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From Non-Negative to General Operator Cost Partitioning. In *Proc. of 29th AAAI Conference on Artificial Intelligence*, 3335–3341. [Cited on page 89.]

Prates, M.; Avelar, P. H. C.; Lemos, H.; Lamb, L. C.; and Vardi, M. Y. 2019. Learning to solve NP-complete problems: a graph neural network for decision TSP. In *Proc. of 33rd AAAI Conf. on Artificial Intelligence (AAAI)*. [Cited on page 20.]

Richter, S.; and Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. In *Proc. of 19th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 273–280. [Cited on pages 12 and 13.]

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *J. Artif. Intell. Res.*, 39: 127–177. [Cited on pages 10, 51, and 91.]

Samadi, M.; Felner, A.; and Schaeffer, J. 2008. Learning from Multiple Heuristics. In aaa (2008), 357–362. [Cited on page 88.]

Scala, E.; Haslum, P.; Thiebaux, S.; and Ramirez, M. 2016. Interval-Based Relaxation for General Numeric Planning. In *Proc. of 22nd European Conference on Artificial Intelligence (ECAI)*, 655–663. [Cited on page 1.]

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1): 61–80. [Cited on page 20.]

Seipp, J. 2023. Scorpion 2023. In *IPC-10 Planner Abstracts*. [Cited on pages 10 and 51.]

Seipp, J.; and Helmert, M. 2013. Counterexample-guided Cartesian Abstraction Refinement. In *Proc. of 23rd Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 347–351. [Cited on page 16.]

Seipp, J.; Pommerening, F.; Röger, G.; and Helmert, M. 2016. Correlation Complexity of Classical Planning Domains. In ijc (2016), 3242–3250. [Cited on page 89.]

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proc. of 30th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 574–584. [Cited on pages 2, 9, and 88.]

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587): 484–489. [Cited on page 1.]

Silver, T.; Chitnis, R.; Curtis, A.; Tenenbaum, J. B.; Lazano-Pérez, T.; and Kaelbling, L. P. 2021. Planning with Learned Object Importance in Large Problem Instances with Graph Neural Networks. In *Proc. of 35th AAAI Conf. on Artificial Intelligence (AAAI)*. [Cited on page 90.]

Slaney, J.; and Thiébaux, S. 2001. Blocks World revisited. *Artif. Intell.*, 125(1–2): 119–153. [Cited on pages xi, xii, 9, 57, and 58.]

Snoek, J.; Larochelle, H.; and Adams, R. P. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems 25 (NIPS)*. [Cited on page 56.]

Speck, D.; and Gnad, D. 2024. Decoupled Search for the Masses: A Novel Task Transformation for Classical Planning. In ica (2024), 546–554. [Cited on page 92.]

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022a. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In ica (2022), 629–637. [Cited on pages 2, 90, and 97.]

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022b. Learning Generalized Policies without Supervision Using GNNs. In *Proc. of 19th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, 474–483. [Cited on pages 2 and 9.]

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022c. Learning Generalized Policies without Supervision Using GNNs. In *ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*. [Cited on page 90.]

Stern, R.; and Juba, B. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *Proc. of 26th Int. Joint Conf. on Artificial Intelligence (IJCAI)*. [Cited on page 91.]

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fišer, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; Segovia-Aguas, J.; and Seipp, J. 2024. The 2023 International Planning Competition. *AI Mag.*, 1–17. [Cited on pages 2, 3, 10, 56, 65, and 90.]

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *J. Artif. Intell. Res.*, 68: 1–68. [Cited on pages 2, 9, and 90.]

Trevizan, F.; Thiébaux, S.; Santana, P.; and Williams, B. 2016. Heuristic Search in Dual Space for Constrained Stochastic Shortest Path Problems. In *Proc. of 26th Int. Conf. on Automated Planning and Scheduling (ICAPS)*. [Cited on page 1.]

Wang, R. X.; and Thiébaux, S. 2024. Learning Generalised Policies for Numeric Planning. In *Proc. of 34th Int. Conf. on Automated Planning and Scheduling (ICAPS)*. [Cited on pages 2, 90, and 100.]

Wehrle, M.; and Helmert, M. 2012. About Partial Order Reduction in Planning and Computer Aided Verification. In ica (2012), 297–305. [Cited on page 92.]

Wehrle, M.; and Helmert, M. 2014. Efficient Stubborn Sets: Generalized Algorithms and Selection Strategies. In *Proc. of 24th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 323–331. [Cited on page 92.]

Xi, K.; Gould, S.; and Thiébaux, S. 2024. Neuro-Symbolic Learning of Lifted Action Models from Visual Traces. In *Proc. of 34th Int. Conf. on Automated Planning and Scheduling (ICAPS)*. [Cited on pages 61 and 91.]

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning Control Knowledge for Forward Search Planning. *J. Mach. Learn. Res.*, 9: 683–718. [Cited on page 88.]

Zhou, J.; Cui, G.; Zhang, Z.; Yang, C.; Liu, Z.; and Sun, M. 2020. Graph Neural Networks: A Review of Methods and Applications. *arXiv preprint arXiv:1812.08434*. [Cited on page 20.]