Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2025)

Leveraging Action Relational Structures for Integrated Learning and Planning

Ryan Xiao Wang, Felipe Trevizan

School of Computing
Australian National University
{ryan.wang, felipe.trevizan}@anu.edu.au

Abstract

Recent advances in planning have explored using learning methods to help planning. However, little attention has been given to adapting search algorithms to work better with learning systems. In this paper, we introduce partial-space search, a new search space for classical planning that leverages the relational structure of actions given by PDDL action schemas – a structure overlooked by traditional planning approaches. Partial-space search provides a more granular view of the search space and allows earlier pruning of poor actions compared to state-space search. To guide partial-space search, we introduce action set heuristics that evaluate sets of actions in a state. We describe how to automatically convert existing heuristics into action set heuristics. We also train action set heuristics from scratch using large training datasets from partial-space search. Our new planner, LazyLifted, exploits our better integrated search and learning heuristics and outperforms the state-of-the-art ML-based heuristic on IPC 2023 learning track (LT) benchmarks. We also show the efficiency of LazyLifted on high-branching factor tasks and show that it surpasses LAMA in the combined IPC 2023 LT and high-branching factor benchmarks.

Extended version — https://arxiv.org/abs/2504.20318
Code — https://doi.org/10.5281/zenodo.15302015

1 Introduction

Development in machine learning has led to significant interest in learning for planning in recent years. The focus of existing work has been on learning domain knowledge automatically in a domain-independent manner to help planning systems. Such domain knowledge can be in the form of generalised policies (Toyer et al. 2018, 2020; Ståhlberg, Bonet, and Geffner 2022a,b, 2023), subgoal structures (Drexler, Seipp, and Geffner 2024; Bonet and Geffner 2024), and heuristics that guide search algorithms. We focus on the common and successful approach of learning heuristics. Existing works have learned heuristics using neural networks (Shen, Trevizan, and Thiébaux 2020; Karia and Srivastava 2021; Chen, Thiébaux, and Trevizan 2024), as well as classical machine learning and optimisation methods (Francès et al. 2019; Chen, Trevizan, and Thiébaux 2024), and have explored learning heuristics to estimate metrics other than cost-to-goal (Ferber et al. 2022; Chrestien et al. 2023; Hao et al. 2024). However, to our knowledge, no work has explored how search algorithms can be adapted to work better with learned heuristics. Specifically, all previously mentioned learning for planning approaches use state-space search, where heuristics are functions of planning states. Little discussion has been given to if this state-based interface between heuristics and search is the most effective, particularly for learning-based heuristics.

In this paper, we introduce partial-space search, a more granular search space than state-space search that takes into account the actions schemas given by the PDDL domain description (Haslum et al. 2019). Action schemas define a natural tree structure of the space of partially instantiated actions, which we call partial actions. Partial-space search is a refinement of state space search that explores this tree structure for more granularity. Searching in this space represents instantiating action schemas one parameter at a time, and transitioning between states when all parameters are instantiated. Partial-space search also represents a change in the interface between heuristics and search, where heuristics are functions of state and partial action pairs.

Partial-space search offers both general advantages and learning specific advantages. The added granularity makes heuristic search more efficient by dividing a single state expansion in state-space search into multiple smaller expansion steps with lower branching factors. This is particularly useful for tasks with high branching factors, regardless of if learned heuristics are used. For learning, we show this additional granularity allows the generation of larger training datasets from the same training tasks and plans, which can be used to train more powerful heuristics. Moreover, designing heuristics require a trade-off between evaluation speed and heuristic informedness. As we will discuss, partial-space search shifts this trade-off towards favouring informedness. This benefits learning-based heuristics, in particular as they continue to become more accurate and potentially slower to evaluate. Ultimately, these advantages mean that partial-space search is potentially better suited for future learning-based heuristics.

To construct heuristics for partial-space search, we view partial actions as sets of actions, and define action set heuristics as functions of state and action sets pairs. To make

partial-space search compatible with any existing state space heuristics, we show how to translate them into action set heuristics. Since our focus is on learning-based heuristics, we also introduce two novel graph representations for state and action set pairs, and extend the methods proposed in Chen, Trevizan, and Thiébaux (2024); Hao et al. (2024); Chen and Thiébaux (2024) to learn action set heuristics from our graph representations.

To evaluate our approach, we implement our contributions in a new planner called LazyLifted. We use the International Planning Competition 2023 learning track benchmarks and additional high branching factor benchmarks. When using the translated action set heuristics (specifically, h^{FF}), partial-space search outperforms state-space search with the original heuristic under high branching factors. When using the learned action set heuristics, LazyLifted outperforms the state-of-the-art learned heuristic. Altogether, LazyLifted outperforms LAMA on the combined benchmarks in terms of coverage.

2 Background

Planning A classical planning problem is a pair $\Pi = \langle D, I\rangle$ of a *domain* D and *instance* I (Geffner and Bonet 2013; Haslum et al. 2019). The domain $D = \langle \mathcal{P}, \mathcal{A}\rangle$ provides the high level structure of the problem by defining a set of *predicates* $\mathcal{P}$ and a set of *action schemas* $\mathcal{A}$. The instance $I = \langle \mathcal{O}, s_0, G\rangle$ provides the task specific information by defining a set of *objects* $\mathcal{O}$, the *initial state* s_0, and the *goal* G. Each predicate $P \in \mathcal{P}$ can be instantiated with objects $o_1, \ldots, o_k \in \mathcal{O}$ to form a ground *atom* $P(o_1, \ldots, o_k)$, where the arity k depends on P. Each action schema $A \in \mathcal{A}$ has *schema arguments* $\Delta(A)$, *preconditions* $\mathrm{pre}(A)$, *add effects* $\mathrm{add}(A)$, and *delete effects* $\mathrm{del}(A)$. Preconditions, add effects, and delete effects are expressed as predicates instantiated with $\Delta(A)$. An action schema can be instantiated with objects, resulting in a ground action where the schema arguments are replaced with the instantiating objects. We denote the set of ground actions with $\mathbb{A}$. A predicate is *static* if it does not appear in the effect of any action schemas.

The problem Π encodes a finite *state space* $\mathbb{S}$, where each state is a set of ground atoms, representing those that are true in the state. The initial state s_0 is an element of $\mathbb{S}$. The goal G is a set of ground atoms, and a state s is a *goal state* if $G \subseteq s$. The ground actions $\mathbb{A}$ define a transition system on $\mathbb{S}$, where each action $a \in \mathbb{A}$ can be applied in any state s with $\mathrm{pre}(a) \subseteq s$, resulting in a new state $(s \setminus \mathrm{del}(a)) \cup \mathrm{add}(a)$. The set of applicable actions in a state s is denoted $\mathbb{A}_s$. The solution to a planning problem is a *plan* – a finite sequence of ground actions that can be applied sequentially starting from the initial state, whose sequential application leads to a goal state. We assume all actions have unit cost, and the cost of a plan is simply its length.

State-space search The most prevalent approach to classical planning is heuristic search on the state space. A (state space) *heuristic* $h : \mathbb{S} \to \mathbb{R}_{\geq 0}$ estimates the quality of a state, with lower values being better. A search algorithm, such as A* or Greedy Best First Search (GBFS), explores the state space under the guidance of the heuristic. This *state-space search* typically requires grounding the planning task, i.e., computing the set of all reachable ground atoms and actions (Helmert 2006). Tasks where this is hard or infeasible are called *hard to ground*. Lifted planners, that do not require grounding the entire task, have been developed to address this issue (Corrêa et al. 2020). Computation of various heuristics for lifted planners can be harder than for grounded planners, but recent works have shown that well-known heuristics can be computed efficiently (Corrêa et al. 2021, 2022).

Learning for Planning via Graphs The current state-of-the-art method for learning heuristics uses the Weisfeiler-Lehman (WL) algorithm. The WL algorithm is an iterative algorithm that computes feature vectors representing the frequency of substructures in a graph. The number of iteration is a hyperparameter that determines the complexity of the substructures. Chen, Trevizan, and Thiébaux (2024) used the WL algorithm to map graph representations to feature vectors and then learn a linear function of these features. The graph representation they used is the *Instance Learning Graph* (ILG). Given a state s and a planning problem Π, the ILG is a graph $G = \langle V, E, c, l\rangle$ where

- Vertices $V = \mathcal{O} \cup s \cup G$.
- Edges $E = \bigcup_{p=P(o_1,\ldots,o_k)\in s\cup G} \{\langle p, o_1\rangle, \ldots, \langle p, o_k\rangle\}$.
- Vertex colours $c(v) \in (\{\mathrm{ag}, \mathrm{ap}, \mathrm{ug}\} \times \mathcal{P}) \cup \{\mathrm{ob}\}$ with

$$v \mapsto \begin{cases} \mathrm{ob}, & \text{if } v \in \mathcal{O} \\ (\mathrm{ag}, P), & \text{if } v = P(o_1, \ldots, o_k) \in s \cap G \\ (\mathrm{ap}, P), & \text{if } v = P(o_1, \ldots, o_k) \in s \setminus G \\ (\mathrm{ug}, P), & \text{if } v = P(o_1, \ldots, o_k) \in G \setminus s \end{cases}$$

- Edge labels $l(e) = i$ for $e = \langle p, o_i\rangle$.

States can be mapped to feature vectors by applying the WL algorithm to their ILG representations. Given small example instances on the same domain with example plans, Chen, Trevizan, and Thiébaux (2024) trained machine learning models to map feature vectors of states in the example plans to their distance to goal in the example plan. They developed the WL-GOOSE system, which uses the trained models as state-of-the-art learned heuristics to guide GBFS in state-space search. Note that in their work, all atoms whose predicates are static are ignored when mapping states to their ILG.

3 Partial-space Search

State-space search ignores the inherent hierarchy of decisions induced by action schemas and their instantiation process. For example, in a planning task of making and serving sandwiches, humans may first decide whether to make or serve a sandwich, and then decide what type of sandwich or who to serve the sandwich to. In state-space search, the planner would typically consider all the sandwich making and sandwich serving actions at the same time, rather than deciding in a hierarchical fashion. To exploit this hierarchy, we introduce partial actions.

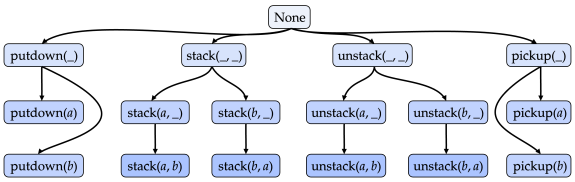

Figure 1: Partial action tree for a Blocksworld task with objects a and b. Darker node colours indicate higher specificity.

Definition 1. *A* partial action *is an action schema with some (incl. none and all) of its schema arguments instantiated by objects. It is denoted as $A(o, \_)$ where $A \in \mathcal{A}$, $o \in \mathcal{O}$, and $\_$ indicates an uninstantiated schema argument. The special partial action* None *indicates that not even the action schema itself specified. For a partial action ρ with k schema arguments instantiated, its* specificity *is* $\mathrm{spec}(\rho) = k + 1$*, where* $\mathrm{spec}(\mathrm{None}) = 0$.

We assume schema arguments of action schemas have a fixed order, and that partial actions have a prefix of the ordered schema arguments instantiated. This way, the partial actions form a *partial action tree* rooted at None, whose children are the action schemas, and whose children are partial actions with specificity 2, etc. (see Figure 1). A partial action ρ is a representation of the set of ground actions in its subtree, denoted $\mathbb{A}^\rho$. We use $\mathbb{A}^\rho_s$ to denote the set $\mathbb{A}^\rho \cap \mathbb{A}_s$. A partial action is applicable in a state s if $\mathbb{A}^\rho_s$ is not empty. Through the partial action tree, partial actions naturally represent the hierarchical nature of actions, which allows defining partial-space search.

Definition 2. *Given a classical planning task Π, the* partial-space search *of Π is a search problem whose search nodes have the form $\langle s, \rho \rangle$ where $s \in \mathbb{S}$ and ρ is a partial action. The search starts at the root node $\langle s_0, \mathrm{None} \rangle$. The successor nodes of each node $\langle s, \rho \rangle$ is given by*

- *If ρ is not fully instantiated, the successors are the nodes with state s and partial action ρ', where ρ' is an applicable child of ρ in the partial action tree.*
- *Otherwise, the successor is $\langle s', \mathrm{None} \rangle$, where s' is the resulting state of applying ρ in s.*

Partial-space search finds a plan for Π by reaching a node $\langle s, \mathrm{None} \rangle$ where s is a goal state. The plan is the sequence of fully instantiated partial actions from the root to this node. Moreover, in practice many search nodes have only one successor node. Such nodes are repeatedly expanded till obtaining multiple successor nodes. The following theorem states the correctness of partial-space search.

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

Proof. Given a planning task Π and let π be a plan found by partial-space search, we first show soundness, i.e., that π is a valid plan. Partial-space search found π by traversing a sequence of nodes $\langle s_0, \rho_0 \rangle, \langle s_1, \rho_1 \rangle, \ldots, \langle s_n, \rho_n \rangle$ where $s_0 = s_0$, $\rho_n = \mathrm{None}$, and s_n is a goal state. For each i, the definition of partial-space search guarantees that ρ_i is an applicable partial action in s_i. Moreover, if ρ_i is fully-instantiated then s_{i+1} is the resulting state of applying ρ_i in s_i, and otherwise the state is unchanged from one node to the next, i.e., $s_i = s_{i+1}$. Given these, since the plan π is obtained by only keeping the fully-instantiated ρ_i's, it must be a valid plan for Π.

Next, we show completeness, i.e., that partial-space search can always find a plan for Π as long as a valid plan exists. Let π be such a valid plan for Π. Each action a in π can be decomposed into a sequence of partial actions with increasing specificity, starting at None and ending at a. Combining these partial actions with the state that a was applied in yields a sequence of partial space search nodes. Doing this for all actions in π yields a sequence of partial space search nodes that are in the search tree of partial-space search. Therefore, partial-space search will find π. □

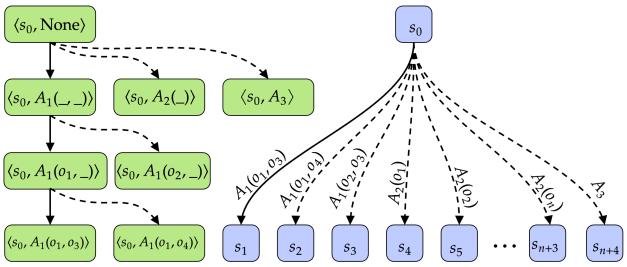

Figure 2: Example expansion trees of partial-space search (left) and state space search (right). Solid lines indicate expanded nodes, dashed lines indicate unexpanded nodes. Expansion decisions are guided by an informed heuristic.

Partial-space search decomposes the process of expanding a single state in state space search into multiple smaller expansion steps that are akin to gradually narrowing down from the set of applicable actions. This can factorise the branching factor of state-space search, as shown in the example in Figure 2. Here, partial-space search removes the need to evaluate all successor states resulting from the application of the A_2 action schema because it determines A_2 itself to be a bad choice. This way, partial space search achieves the same outcome as state-space search but with fewer heuristic evaluations, which is particularly useful when branching factors are high or the heuristic is expensive to evaluate but very accurate. The latter case is what we mean by partial-space search moving the trade-off between heuristic evaluation speed and informedness towards informedness. This shift is particularly relevant for learning-based heuristics. Machine learning, particularly deep learning, has trended towards more powerful models that are more expensive to compute. Partial-space search is more suited for this trend. Beyond being more suited for applying learning-based heuristics, partial-space search also allows generating larger training datasets due to the increased granularity. We discuss this further once we introduce how we generate training data using partial-space search.

4 Action Set Heuristics

In order to guide partial-space search, we need heuristics that evaluate search nodes of partial-space search, which are state and partial action pairs. Given such a pair $\langle s, \rho \rangle$, we define heuristics by viewing ρ as the set of applicable actions $\mathbb{A}_s^\rho$. This is a more general form of heuristics than those we need, but helps simplify our discussion.

Definition 3. *An* action set heuristic *is a function* $h : \mathbb{S} \times 2^{\mathbb{A}} \to \mathbb{R}$. *Given a partial action* ρ*, we will use* $h(s, \rho)$ *to denote* $h(s, \mathbb{A}_s^\rho)$.

For a classical planning task, action set heuristics can be used for heuristic search over the partial state space. Moreover, given an action set heuristic h, we can obtain a state space heuristic h' with $s \mapsto h(s, \mathrm{None})$. Next, we describe two ways to obtain action set heuristics.

4.1 Restriction Heuristics

Our first method is to translate existing state space heuristics to action set heuristics. Given a state space heuristic h, a state s, and a set of actions Λ, we obtain an action set heuristic by modifying the original problem such that only actions in Λ can be applied in s, then computing $h(s)$ in this modified problem. We achieve this by adding a predicate ϵ to the problem to indicate if an action in Λ has been applied. We then modify the actions such that all actions add ϵ and only actions in Λ can be applied without ϵ. More formally:

Definition 4 (Restriction heuristic)**.** *For a planning task* Π*, given a state space heuristic* h*, a state* s*, and a set of actions* Λ*, the* Λ-restricted task Π_Λ *is* Π *with the following modifications applied sequentially:*

1. *an additional predicate* ϵ*, which has no parameters and is initially false, is added;*
2. *the predicate* ϵ *is added as a precondition to all action schemas;*
3. *each* $a \in \Lambda$ *is added as an action schema with no schema arguments that is identical to* a*, except with the additional add effect* ϵ*;*

The restriction heuristic $h_{\mathrm{rs}}(s, \Lambda)$ *is the action set heuristic that takes the value of* $h(s)$ *computed for the task* Π_Λ.

Despite its generality, computing restrictions heuristics by definition can be computationally expensive. This is because the task Π_Λ must be constructed for each possible action set Λ, which can be exponential in the number of actions. However, for some heuristics, it is possible to overcome this by not explicitly computing the restricted task.

An example heuristic whose restricted version can be efficiently computed is the lifted version of the h^{FF} heuristic (Corrêa et al. 2022). Here, we only outline the necessary adaptations for computing its restriction heuristic and more details can be found in Wang and Trevizan (2025b). Corrêa et al. (2022) utilised Datalog rules corresponding to action schemas, and required preprocessing the task to generate these rules. For the restriction heuristic of h^{FF}, namely $h^{\mathrm{FF}}_{\mathrm{rs}}$, we apply modifications 1 and 2 (Def. 4) to the task, which are independent of the action set Λ, and then preprocess the modified task as described in their work. After, for each heuristic evaluation, we apply modification 3 by generating temporary Datalog rules corresponding to each action in Λ, and then perform the heuristic computation as in their work. This modification is temporary, i.e., it is discarded after the heuristic evaluation.

4.2 Graph Representations

Restriction heuristics represent a general approach to obtaining action set heuristics from existing state space heuristics. An alternative is to learn action set heuristics from scratch, which yields heuristics dedicated for partial-space search instead of merely adapted state space heuristics. Recent works on learning heuristics for planning have not only shown that very powerful heuristics can be learned, but also that heuristic learning is very flexible. To learn a heuristic using the WL algorithm, one only needs to define a graph representation for search nodes and then specify through training dataset set what properties the heuristic should have. There is no need to define the heuristic function itself, and even better, the learned WL heuristics can also be interpreted (Chen, Trevizan, and Thiébaux 2024).

To learn action set heuristics, we start by defining two novel graph representations of state and action set pairs. These two graphs represent shallow and deep embeddings of the action set into the ILG of the state, respectively. They are used to generate feature vectors of state and action pairs through the WL algorithm.

As a minor improvement over the ILG, we encode in the object colours the static predicates with arity 1 that apply to each object. That is, like the ILG, both of graphs will include object nodes. We use $\mathcal{P}_o$ to denote the set of static predicates of arity 1 such that for each $P \in \mathcal{P}_o$, $P(o)$ holds in the initial state (and hence any state reachable from it). The colour of the object node o in our graphs is $\mathcal{P}_o$. This allows our graphs to consider some static predicates, unlike the ILG that ignores all of them.

Action-Object-Atom Graph Our first graph, called the Action-Object-Atom Graph (AOAG, Def. 5), is an extension of the ILG that directly encodes the actions in the action set as nodes in the graph.

Definition 5 (AOAG)**.** *Given a state* s *and a set of actions* Λ*, if* $\mathbb{A}_s \subseteq \Lambda$*, then the* ***Action-Object-Atom Graph*** *(AOAG) of* (s, Λ) *is simply the ILG of* s *with static object colours. This in particularly happens when the partial action inducing* Λ *is* None*. Moreover, if* $\Lambda = \{a\}$*, then the AOAG is the ILG of the state obtained by applying* a *in* s*, again with static object colours.*

Otherwise, let the ILG of s *with static object colours be* $\langle V_{\mathrm{ILG}}, E_{\mathrm{ILG}}, c_{\mathrm{ILG}}, l_{\mathrm{ILG}} \rangle$*, the AOAG is* $\langle V, E, c, l \rangle$*, where*

- $V = V_{\mathrm{ILG}} \cup \Lambda$
- $E = E_{\mathrm{ILG}} \cup \bigcup_{a=A(o_1,\ldots,o_k)\in\Lambda} \{\langle a, o_1 \rangle, \ldots, \langle a, o_k \rangle\}$
- $c(u)$ *is* $c_{\mathrm{ILG}}(u)$ *if* $u \notin \Lambda$*, otherwise for* $u = A(\ldots) \in \Lambda$ *we have* $c(u) = A$.
- $l(e)$ *is* $l_{\mathrm{ILG}}(e)$ *if* $e \in E_{\mathrm{ILG}}$*, otherwise for* $e = \langle a, o_i \rangle$ *we have* $l(e) = i$.

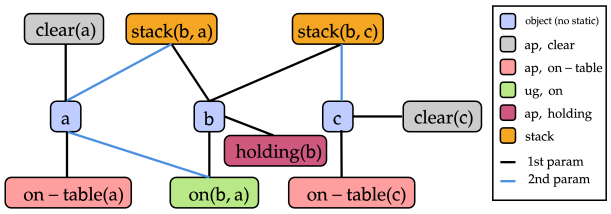

Figure 3: AOAG (Def. 5) of a Blocksworld Instance with blocks a and c on the table and b being held, here the goal is to place b on a and Λ contains all applicable `stack` actions.

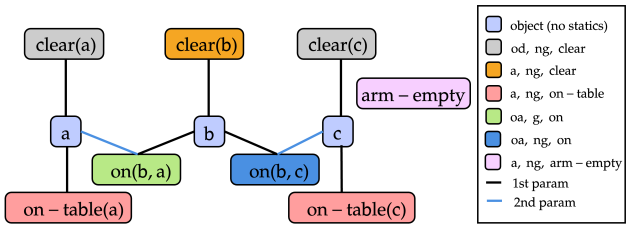

Figure 4: AEG (Def. 6) of a Blocksworld Instance with blocks a and c on the table and b being held, with the goal to place b on a and Λ containing all applicable `stack` actions.

AOAG is a shallow embedding of the action set into the ILG of the state as it only introduces vertices representing actions and edges connecting actions to their parameters without representing the actions' effects. An example of AOAG for Blocksworld is shown in Figure 3.

Action Effect Graph An alternative to directly encoding the actions in the graph is to instead analyse the effects of the actions in Λ. This is the approach taken to construct the Action Effect Graph (AEG, Def. 6), which is a deep embedding of action sets into ILGs of the states. Here, we categorise the action effects into two types, *unavoidable* and *optional*, and represent them in the graph. Unavoidable effects are those that are present in all actions in Λ, which we can directly apply in the state s. Optional effects are those that are present in some actions in Λ but not all, which we can choose to apply or not. Since our action sets are induced from partial actions, unavoidable effects are those that only depend on the schema arguments instantiated so far.

Definition 6 (AEG)**.** *Given a state s and a set of actions Λ. If $\mathbb{A}_s \subseteq \Lambda$, the set $\mathrm{unav}^+(\Lambda)$ denotes the* unavoidable add effects *of Λ, i.e. $\bigcap_{a\in\Lambda} \mathrm{add}(a)$; the set $\mathrm{unav}^-(\Lambda)$ denotes the* unavoidable delete effects *of Λ, i.e. $\bigcap_{a\in\Lambda} \mathrm{del}(a)$; the set $\mathrm{opt}^+(\Lambda)$ denotes the* optional add effects *of Λi, i.e. $\bigcup_{a\in\Lambda} \mathrm{add}(a) \setminus \mathrm{unav}^+(\Lambda)$; and lastly the set $\mathrm{opt}^-(\Lambda)$ denotes the* optional delete effects *of Λ, i.e. $\bigcup_{a\in\Lambda} \mathrm{del}(a) \setminus \mathrm{unav}^-(\Lambda)$. As a special case, if $\Lambda = \mathbb{A}_s$, then all these sets are empty. This in particular happens when the partial action inducing Λ is* None*.*

*Let s' be the state given by applying the unavoidable effects in s, i.e. $(s \setminus \mathrm{unav}^-(\Lambda)) \cup \mathrm{unav}^+(\Lambda)$. The **Action Effect Graph** (AEG) is $\langle V, E, c, l\rangle$ where*

- $V = \mathcal{O} \cup G \cup s' \cup \mathrm{opt}^+(A) \cup \mathrm{opt}^-(A)$
- $E = \bigcup_{p=P(o_1,\dots o_k)\in V\setminus\mathcal{O}} \{\langle p, o_1\rangle \dots, \langle p, o_k\rangle\}$
- $c : V \to 2^{\mathcal{P}} \cup (\{\mathrm{a}, \mathrm{u}, \mathrm{oa}, \mathrm{od}\} \times \{\mathrm{g}, \mathrm{ng}\} \times \mathcal{P})$. *For $o \in \mathcal{O}$, $c(o) = \mathcal{P}_o$. Otherwise, for $p = P(o_1, \dots, o_k) \in V \setminus \mathcal{O}$, $c(p) = (\alpha, \beta, P)$ where*

$$\alpha = \begin{cases} \mathrm{oa}, & \text{if } p \in \mathrm{opt}^+(A) \\ \mathrm{od}, & \text{if } p \in \mathrm{opt}^-(A) \\ \mathrm{u}, & \text{if } p \in G \setminus (s' \cup \mathrm{opt}^+(A)) \\ \mathrm{a}, & \text{otherwise} \end{cases}$$

and β is g *if $u \in G$ and* ng *otherwise. Here* a, u, oa, od *mean **a**chieved, **u**nachieved, **o**ptional **a**dd, and **o**ptional **d**elete respectively,* g *and* ng *mean **g**oal and **n**on-**g**oal respectively.*

- $l : E \mapsto \mathbb{N}$ *with* $\langle p, o_i\rangle \mapsto i$.

Unlike the AOAG, the AEG is a deep embedding of the action set into the ILG of the state – it considers the effects of the actions in the action set, and represents them in the graph to reflect how they change the state. An example of AEG for Blocksworld is shown in Figure 4.

4.3 Training Action Set Heuristics

We assume that our training data consists of a set classical planning tasks $\Pi_1, \dots, \Pi_n$ that share the same planning domain, with corresponding plans $\pi_1, \dots, \pi_n$. Each plan is a sequence of actions denoted $\pi_i = a_1, \dots, a_{|\pi_i|}$, which starts at the initial state s_0 and visits the states $s_1, \dots, s_{|\pi_i|}$. Each such plan represents a sequence of steps in state-space search. For partial-space search, we decompose each plan action into a sequence of partial actions of increasing specificity starting at None, and pair them with the corresponding state that they are applied in. This gives a sequence of state and partial action sequence pairs $\langle s_0, (\rho_{1,1}, \dots, \rho_{1,n_1})\rangle, \dots, \langle s_{|\pi_i|-1}, (\rho_{|\pi_i|,1}, \dots, \rho_{|\pi_i|,n_{|\pi_i|}})\rangle$, where $\rho_{j,1}, \dots, \rho_{j,n_j}$ are the partial actions obtained from decomposing a_j. Note that $\rho_{j,1} = \mathrm{None}$ and ρ_{j,n_j} is always a fully-instantiated action, with s_{j+1} the resulting state of applying it in s_j. Given this, we learn action set heuristics through ranking by extending the methods proposed in Chen and Thiébaux (2024). We have also explored doing so through regression in a similar way to what Chen and Thiébaux (2024) did, but found poor performance compared to ranking. This matches the findings of various works, including Garrett, Kaelbling, and Lozano-Pérez (2016); Chrestien et al. (2023), and Hao et al. (2024).

Dataset Generation We use ϕ to denote the function that maps state and partial action pairs to feature vectors through either AOAG or AEG using the WL algorithm. Given a state and partial action sequence pair $\langle s, (\rho_1, \dots, \rho_n)\rangle$, we produce a dataset consisting of tuples of the form $\langle \mathbf{x}, \mathbf{x}', \delta, \sigma\rangle$, where $\mathbf{x}$ and $\mathbf{x}'$ are feature vectors, δ is the desired difference between their heuristic values, and σ is the importance. Each such tuple is a ranking relation. It specifies that the heuristic should assign a value to $\mathbf{x}$ that is at least δ lower than the value assigned to $\mathbf{x}'$. The value σ indicates how important it is for the heuristic to satisfy this ranking relation. Note

that the importance weights σ did not appear in Chen and Thiébaux (2024), but we found it useful to balance different types of tuples. There are four types of tuples described below, we distinguish importance of tuples by the type of the tuple only and not by the specific tuple itself.

1. *Layer predecessors*, which rank later partial actions better than earlier partial actions:

$$\{\langle \phi(s, \rho_1), \phi(s', a'), 1, \sigma_{\text{lp}} \rangle\}$$
$$\cup \{\langle \phi(s, \rho_{i+1}), \phi(s, \rho_i), 1, \sigma_{\text{lp}} \rangle \mid 1 \leq i < n\}$$

   where s' is the state before s in the training plan, and a' is the action that transitioned s' to s.

2. *Layer siblings*, which rank partial actions better than those with the same specificity but not in the plan-induced sequence. Specifically, for each ρ_i in the sequence and for each partial action ρ' applicable in s but not in the sequence with $\text{spec}(\rho_i) = \text{spec}(\rho')$, we produce the tuple $\langle \phi(s, \rho_i), \phi(s, \rho'), 0, \sigma_{\text{ls}} \rangle$.

3. *State predecessors*, which rank partial actions in the plan-induced sequence better than the state s. For each ρ_i, we produce the tuple $\langle \phi(s, \rho_i), \phi(s, \text{None}), 1, \sigma_{\text{sp}} \rangle$.

4. *State siblings*, which ranks ρ_n better than other fully instantiated partial actions not in the sequence. Specifically, for each fully instantiated applicable partial action $\rho' \neq \rho_n$, we produce the tuple $\langle \phi(s, \rho_n), \phi(s, \rho'), 0, \sigma_{\text{ss}} \rangle$.

Here, σ_{lp}, σ_{ls}, σ_{sp}, and σ_{ss} are all hyperparameters. Layer predecessors establish a chain of ranking relations throughout each training plan. State predecessors act as fast tracks for such relations, similar in spirit to skip connections in neural networks. Layer siblings ensure that partial actions in the sequence are preferred. State siblings emphasising this for fully instantiated actions. The δ difference values for the predecessors tuples are 1 to indicate that successors are strictly better, while for siblings tuples it is 0 to indicate that the siblings may be equally good.

We previously stated that our dataset generation method using partial-space search yields more data than state-space search. Consider a hypothetical planning task with α action schemas, each with k parameters, and β objects that can instantiate any parameter of any schema independently of other parameters. For a plan of length n, the state-of-the-art method by Hao et al. (2024) generates $n\alpha\beta^k$ tuples. Our method generates $2n(k+1)+n(\alpha\beta^k-1)+n\sum_{i=0}^{k}(\alpha\beta^i-1)$ tuples, which is approximately $n(k+2\alpha\beta^k)$ tuples. Furthermore, our method not only generates more data but our data is more granular, i.e., with a higher ratio of optimal decisions to available decision. Specifically, Hao et al. (2024)'s dataset encodes n optimal actions resulting in a ratio of $1/(\alpha\beta^k)$, i.e., the correct action choice is always 1 out of the branching factor of the grounded actions $\alpha\beta^k = |\mathbb{A}|$. In contrast, our method encodes $n(k+1)$ optimal decisions (i.e., optimal action schema, optimal first parameter, etc.) resulting in an approximate density of $(k+1)/(k+2\alpha\beta^k)$. For $\alpha\beta^k(k-1) > k$, our dataset is denser, i.e., offers more optimal decisions to guide the learning process. Note that this condition almost always holds since $\alpha\beta^k = |\mathbb{A}| \gg 2$.

Ranking Model Our complete dataset is the collection of all tuples from all state and partial action sequence pairs, namely $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{x'}_i, \delta_i, \sigma_i) \mid i \in [|\mathcal{D}|]\}$. Once having generated this, we learn a linear model with weights $\mathbf{w}$ by solving the following linear program,

$$\min_{\mathbf{w} \in \mathbb{R}^n, \mathbf{z} \in \mathbb{R}^{|\mathcal{D}|}_{\geq 0}} \quad C \sum_i \sigma_i \mathbf{z}_i + \|\mathbf{w}\|_1$$
$$\text{s.t.} \quad \mathbf{w}^T(\mathbf{x}_i - \mathbf{x'}_i) \geq \delta_i - \mathbf{z}_i \qquad \forall i$$

Here, C is a regularisation hyperparameter, and the variables $\mathbf{z}$ represent the slack variables for each of the constraints given by the ranking relations. The term $\|\mathbf{w}\|_1$ is not linear itself, but it is trivially encoded using auxiliary variables and linear constraints. The optimisation target represents that, for all i, $\mathbf{w}$ should map $\mathbf{x}_i$ and $\mathbf{x'}_i$ to values that are at least δ_i apart as much as possible according to the importance σ_i (first term), while being subject to regularisation (second term).

In practice, this linear program can be very large. We use constraint and column generations techniques adapted from Dedieu, Mazumder, and Wang (2022) to solve it efficiently. Moreover, we tune the hyperparameter C automatically. Our training instances are ordered roughly by problem size in ascending order. We use the first 80% of instances as our training set and the rest as our validation set. We tune C by applying Bayesian optimisation to minimise the objective value $\sum_i \sigma_i \mathbf{z}_i$ on the validation set.

Once $\mathbf{w}$ is learned, we obtain an action set heuristic $h_{\mathbf{w}}$ that first maps state and action set pairs to feature vectors, then uses the dot product of the feature vectors and $\mathbf{w}$ as the heuristic value.

5 Experimental Evaluation

We implemented partial-space search and action set heuristics in a new planner, LazyLifted, and evaluated it against the state-of-the-art. We test its performance using benchmarks from the International Planning Competition (IPC) 2023 learning track (Taitler et al. 2024) and new high branching factor benchmarks. The IPC set contains 10 domains, including Blocksworld, Ferry, and Transport, among others.

The high branching factor (HBF) set contains 5 domains. It includes a version of Blocksworld, Blocksworld Large, with many blocks that are irrelevant to the goal, which induces a high branching factor. This is adapted from the Blocksworld Large benchmarks by Lauer et al. (2021) to fit the IPC 2023 learning track style. We also include three versions of Transport, Transport Sparse, Transport Dense, and Transport Full, which reflect the underlying graphs are sparse, dense, and fully connected. Branching factor on Transport increases with graph density. Lastly, we introduce a new domain, Warehouse, which models multiple stacks of boxes with some needing to be removed. Warehouse resembles Blocksworld in the modelling of box stacks, but differs as boxes can move in a single action between the tops of two stacks. This results in a high branching factor, quadratic to the number of stacks, which is high even for small instances.

Each IPC or HBF benchmark contains up to 99 training instances and 90 test instances. The test instances are ordered and grow in size rapidly. The training instances are

roughly the same size as the smallest 30 test instances. A detailed description of the benchmarks, including their problem sizes and branching factors, is in Wang and Trevizan (2025b). Our code and benchmarks is available from Wang and Trevizan (2025a).

Our learning-based heuristics require sample plans on training instances. For the IPC set, we use the same training plans as Chen, Trevizan, and Thiébaux (2024), which are generated by the Scorpion optimal planner (Seipp, Keller, and Helmert 2020) under a 30 minutes timeout for each instance. For the HBF set, we generate training plans for the Transport variants using the same method. For Blocksworld Large and Warehouse, Scorpion cannot solve most of the training instances in the time limit, so we use the first plan returned by LAMA (Richter and Westphal 2010) under the same time limit.

LazyLifted has importance hyperparameters for dataset generation (Sec. 4.3). We set these to $\sigma_{\text{lp}} = 0.5$, $\sigma_{\text{ls}} = 2.0$, $\sigma_{\text{sp}} = 0.5$, and $\sigma_{\text{ss}} = 1.0$ when training AOAG based heuristics and to $\sigma_{\text{lp}} = 0.5$, $\sigma_{\text{ls}} = 2.0$, $\sigma_{\text{sp}} = 0.5$, and $\sigma_{\text{ss}} = 0.75$ when training AEG based heuristics. These values were found to work well across all domains except Warehouse. On Warehouse, the high branching factor even on small training instances meant that there are a lot of siblings, which sometimes made the optimal weights close to zero. We use $\sigma_{\text{lp}} = 2.0$, $\sigma_{\text{ls}} = 1.5$, $\sigma_{\text{sp}} = 2.0$, and $\sigma_{\text{ss}} = 0.5$ for Warehouse for both AOAG and AEG. Lastly, we use two iterations of the WL algorithm for training the heuristic.

We compare LazyLifted against state-of-the-art baselines LAMA, Powerlifted (PWL), and WL-GOOSE. LAMA is a strong satisficing planner that uses multiple heuristics with additional optimisation techniques (Richter and Westphal 2010). We use LAMA with its first plan output. Powerlifted is a lifted planner that specialises in solving hard-to-ground planning tasks, which is particularly relevant as many high branching factor tasks are also hard-to-ground, and hence require lifted planning. WL-GOOSE learns state-of-the-art heuristics for state-space search to guide GBFS (Chen, Trevizan, and Thiébaux 2024). We use our own implementation of WL-GOOSE, which performs slightly better than the original version. We trialed several configurations for WL-GOOSE, namely the configuration used in original paper, the recommended configuration the WL-GOOSE codebase[1], and a variation of the recommended configuration that only uses 2 WL iterations instead of 3 to match LazyLifted. Ultimately, we only present results for the last configuration as it has the best overall performance.

We do not consider other learning-based planners as all that we are aware of perform noticeably worse than WL-GOOSE on the IPC set as shown either by Chen, Trevizan, and Thiébaux (2024) or the results in the IPC 2023 learning track (Taitler et al. 2024).

Moreover, since action set heuristics can be used as state-space heuristics, we run all our action set heuristics on both state and partial-space search (denoted $\text{S}^3$and PS^2) to isolate the effect of the search space. This means that we compare both the state-space h^{FF} heuristic (denoted S^3-FF) and the partial-space $h^{\text{FF}}_{\text{rs}}$ heuristic (denoted PS^2-FF). It also means that we train two action set heuristics, one for AOAG and one for AEG, and use them on both search spaces, resulting in four possible configurations.

[1]Available from https://github.com/DillonZChen/goose.

		Baseline				New				
Set	Domain	LAMA	WL-GOOSE	PWL	S^3-FF	S^3-AOAG	S^3-AEG	PS^2-FF	PS^2-AOAG	PS^2-AEG
IPC23 LT	blocksworld	60	77	37	29	89	**90**	24	89	84
	childsnack	35	36	0	16	37	**38**	14	9	15
	ferry	68	**90**	0	60	**90**	88	49	**90**	87
	floortile	**11**	3	9	10	1	1	6	1	1
	miconic	**90**	**90**	81	77	**90**	**90**	68	**90**	**90**
	rovers	**69**	34	53	28	34	32	34	34	26
	satellite	**89**	41	0	49	39	53	50	29	41
	sokoban	**40**	27	34	32	27	27	29	27	24
	spanner	30	71	30	30	71	71	30	**72**	**72**
	transport	**66**	47	53	36	53	50	38	54	54
sum IPC coverage		**558**	516	297	367	531	540	342	495	494
HBF	blocksworld-large	7	31	2	0	35	18	0	**74**	48
	transport-sparse	62	37	58	31	41	37	31	**64**	58
	transport-dense	**66**	57	52	39	59	51	36	57	57
	transport-full	**66**	58	0	42	63	61	41	55	60
	warehouse	30	15	**90**	35	58	27	54	79	49
sum HBF coverage		231	198	202	147	256	194	162	**329**	272
sum total coverage		789	714	499	514	787	734	504	**824**	766

Table 1: 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.

All experiments are conducted on a cluster with Intel Xeon 3.2 GHz CPU cores. Training for both LazyLifted and WL-GOOSE is performed using 1 core with 32 GB of memory with a 12-hour timeout, excluding data generation, i.e., solving training instances. Both data generation and solving testing problems are done using 1 core and 8 GB of memory with a 30-minutes timeout per problem.

We evaluate planners on two metrics, coverage and quality. Coverage is the number of problem solved and the results per domain are shown in Table 1. The quality score of a planner on a task is 0 if it does not find a plan and C^*/C otherwise, where C is the cost of the plan found by the planner and C^* is the cost of the best plan found by any planner in the experiment. Table 2 shows the sum of quality scores per domain. It is worthnoting that, due to the benchmark setup where the instances in each domain scale up in size quickly, even a small increase either metric can represent a significant improvement. See Wang and Trevizan (2025b) for additional results.

Comparison against baselines Overall, our learned action set heuristics outperform WL-GOOSE in both total coverage and total quality scores across both partial and state-space search. On the IPC set, only learned action set heuris-

Set	Domain	Baseline				New				
		LAMA	WL-GOOSE	PWL	S^3-FF	S^3-AOAG	S^3-AEG	PS^2-FF	PS^2-AOAG	PS^2-AEG
IPC23 LT	blocksworld	36	73	21	13	88	**89**	12	88	82
	childsnack	22	36	0	12	37	**38**	8	9	14
	ferry	57	87	0	55	89	86	33	**90**	86
	floortile	**10**	3	9	9	1	1	4	0	1
	miconic	73	**89**	77	76	**89**	**89**	56	82	82
	rovers	**67**	21	52	25	19	17	30	17	13
	satellite	**85**	31	0	47	26	41	43	17	25
	sokoban	**35**	19	29	27	19	19	23	14	12
	spanner	30	67	30	30	67	67	30	68	**68**
	transport	**63**	39	47	29	43	43	26	41	44
sum IPC quality		480	465	264	324	478	**490**	265	426	426
HBF	blocksworld-large	7	22	2	0	27	12	0	**64**	39
	transport-sparse	**59**	31	48	21	27	23	21	47	38
	transport-dense	**63**	48	46	31	50	47	25	47	36
	transport-full	**63**	48	0	39	52	57	32	42	40
	warehouse	30	6	**89**	35	54	14	54	57	17
sum HBF quality		222	154	185	126	209	153	132	**257**	171
sum total quality		**702**	619	449	449	687	643	398	683	597

Table 2: 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.

tics with state-space search (S^3-AOAG/AEG) outperform WL-GOOSE by around 5% in coverage and quality. The improvements are more substantial on the HBF set, where our learned action set heuristics with partial-space search (specifically, PS^2-AOAG) are up to 60% better in coverage and quality scores than WL-GOOSE. In the combined set of problems, S^3-AOAG and PS^2-AOAG obtained 10% to 15% better total coverage and 10% improvement in total quality over WL-GOOSE. Additional results available in Wang and Trevizan (2025b) show that the learned action set heuristics perform similarly to WL-GOOSE on runtime overall, with the comparison varying widely for specific domains. Altogether, given WL-GOOSE's state-of-the-art status in learned heuristics, these results represent a significant improvement.

LazyLifted also outperforms LAMA in several cases, namely coverage and quality for the HBF set, quality for the IPC set, and total coverage when considering the combined set. LazyLifted is notably strong on IPC domains such as Blocksworld, Ferry, and Spanner, and the non-Transport HBF domains, while still lacking behind on domains such as Rovers and Satellite. It is noteworthy that LazyLifted finds low-quality plans on Transport domains compared to coverage, indicating that LazyLifted's learned action set heuristics are not able to outperform LAMA on these domains. In most other cases, LazyLifted finds plans with reasonable quality. However, the large makeup of Transport domains in the benchmark sets means that LazyLifted outperforms LAMA only in total coverage but not total quality.

It is important when interpreting these results to note that LAMA employs various planning techniques, e.g., multi-queue search and preferred operators, while our system only uses a single heuristic with GBFS over either partial or state-space search. In theory, techniques used in LAMA could also be applied in LazyLifted to further enhance performance. We do not use them to isolate the effect of our contributions over existing techniques.

Lastly, LazyLifted performs significantly better than Powerlifted (PWL) on both the IPC set and HBF set. The improvement over Powerlifted on the IPC set is unsurprising given that Powerlifted is not the state-of-the-art in general classical planning. However, given the state-of-the-art nature of Powerlifted in hard-to-ground planning, the improvement over Powerlifted on the HBF set indicates improvements LazyLifted makes in this space.

Partial-space search versus state-space search Our experiments are set up to isolate the effect of partial-space search over state-space search by running the same heuristics on both search spaces. Our results show that partial-space search is not beneficial on the IPC set, as indicated by coverage and quality scores. On the HBF set, partial-space search is unsurprisingly very impactful, as indicated by the substantial improvement in coverage and quality scores.

To further investigate, we compared the branching factor (average number of successors per expansion), number of expansions, and number of evaluations between partial and state-space search (see Wang and Trevizan (2025b) for full results). In terms of median, on the combined IPC and HBF set, partial-space search reduces the branching factor by 88%, increases the number of expansions by around 7 times, and reduces the number of evaluations by approximately 14%. This indicates that partial-space search can reduce the overall work required to solve planning tasks, which is typically dominated by the number of evaluations. These results also demonstrated that the IPC domains tend to have a lower branching factor, leading to more evaluations for partial-space search. This explains why partial-space search is not beneficial on the IPC set.

Informedness of learned action set heuristics Our experiments also highlight the effectiveness of partial-space search at training heuristics. The only difference between WL-GOOSE and S^3-AOAG/AEG is the heuristic used, where the latter uses partial-space search to train action set heuristics, then uses these action set heuristics as regular state-space heuristics. The learned action set heuristics with state-space search generally outperform WL-GOOSE on both coverage and quality. Moreover, they also generally have less search node expansions and heuristic evaluations (see Wang and Trevizan (2025b)). These results overall indicate that training with partial-space search leads to more informed heuristics. We hypothesise that this improvement partially stems from the larger training datasets generated by LazyLifted, which are on average 2.62 times larger than those generated by WL-GOOSE from the same training plans.

AOAG versus AEG Our results show that AEG works slightly better on the IPC set, while AOAG works much better on the HBF set, regardless of the search space. We hypothesise that this is due to the average of size of action sets on these domains and the difference in the way the two graphs encode action sets. AEG encodes action sets more deeply into the graph by considering action effects, but does not label in the graph which optional effect belongs to which actions. This likely works better under low branching factors, as there is less ambiguity between effects of different actions. In contrast, the shallow action encoding of AOAG is more explicit, which likely works better under high branching factors by avoiding this ambiguity.

How well does h^{FF} translate to partial-space search? Another case of interest is the comparison between $h^{\mathrm{FF}}_{\mathrm{rs}}$ (PS^2-FF) and h^{FF} (S^3-FF): $h^{\mathrm{FF}}_{\mathrm{rs}}$ is slightly worse than h^{FF} on the IPC set and slightly better than HBF set. This is consistent with the effects of partial-space search, showing that $h^{\mathrm{FF}}_{\mathrm{rs}}$ is a faithful translation of h^{FF} to partial-space search.

On the HBF Transport variants, why does coverage of partial-space search decrease as graph density increases? We observed that as graph density increases, the branching factor increases, which should favour partial-space search. However, the path-finding problem also becomes much easier, meaning that the high branching factor costs on dense graphs is paid far less often than on sparse graphs, making partial-space search overall less effective.

Why do learned action set heuristics not perform well on domains such as Rovers and Satellite? Our methods for learning action set heuristics are based on those used in WL-GOOSE (Chen, Trevizan, and Thiébaux 2024), namely feature extraction using the WL algorithm, and we inherit weaknesses of their methods. Our learned action set heuristics perform similarly to WL-GOOSE on these domains, indicating that our methods are likely not the cause of the poor performance.

6 Conclusion, Related, and Future Work

We introduced partial-space search for planning, which is a new search space that allows search and learned heuristics to be more deeply integrated. Partial-space search decomposes branching factor, allows more training data to be generated out of the same training plans, and is able to shift the evaluation speed versus informedness trade-off of heuristics to favour informedness. This results in significant empirical improvements, particularly under high branching factors, over existing learned heuristics and even outperforms the state-of-the-art non-learning planner LAMA, which indicates the potential of deep integrations between planning and learning.

Partial-space search requires specialised action set heuristics that evaluate a set of actions on a state. We showed how partial-space search can be generally useful by providing a method to automatically convert existing state-space heuristics to action set heuristics. Moreover, we introduced novel graph representations that encapsulate action set semantics in various ways for learning well-informed action set heuristics. These learned action set heuristics are effective whether employed in partial or state-space search.

In terms of related work, learning for planning, particularly in the form of learned heuristics, has gathered notable research attention. Previous works have developed methods for learning state-space heuristics (Chen, Trevizan, and Thiébaux 2024) and explored possible alternative heuristic targets (Garrett, Kaelbling, and Lozano-Pérez 2016; Ferber et al. 2022; Chrestien et al. 2023; Hao et al. 2024). Yet to our knowledge, this is the first work focused on how to design a search algorithm based on the capabilities of learned heuristics.

For dealing with large branching factors, Yoshizumi, Miura, and Ishida (2000) proposed a variant of the A* search algorithm (PEA*) that partially expands search nodes to reduce the memory cost induced by high branching factors. Goldenberg et al. (2014) extended PEA* to also reduce the time cost of search, but required domain and heuristic knowledge. Their works had focused on modifying the search algorithm, while our partial-space search modifies the search space by incorporating heuristic predictions over sets of actions and are hence complementary.

In the context of Monte-Carlo Tree Search, Shen et al. (2019) explored how to guide search with generalised policies. Their work is similar to our as they also explore alternative forms of heuristics, in their case, a generalised policy that predicates a probability distribution over applicable actions in a state. Similar to how partial-space search introduces a different interface between search and heuristics over the traditional state-based interface, their approach uses a probability distribution as the interface. Their work is focused on the application of search to overcome weaknesses in learned generalised policies, while our work redesigns search itself to better suit learned heuristics. Their work also achieved limited empirical success, while we demonstrated significant empirical improvements over existing learning-based methods.

In future work, we hope to extend the applicability of partial-space search and explore other forms of deep integrations between planning and learning. Extending the applicability involves finding efficient adaptations of existing heuristics and other search techniques to partial-space search. It also involves extending partial-space search to other planning paradigms, such as numeric planning and planning under uncertainty. Exploring other forms of deep integrations involves exploring other ways learning and planning can help each other, such as using learning to guide planners to focus on specific subtasks of planning tasks.

Acknowledgments

The computing resources for this work was supported by the Australian Government through the National Computational Infrastructure (NCI) under the ANU Startup Scheme.

References

Bonet, M.; and Geffner, H. 2024. General Policies, Subgoal Structure, and Planning Width. *J. Artif. Intell. Res.*

Chen, D. Z.; and Thiébaux, S. 2024. Learning Heuristics for Numeric Planning. In *Proc. NeurIPS 2024*.

Chen, D. Z.; Thiébaux, S.; and Trevizan, F. 2024. Learning Domain-Independent Heuristics for Grounded and Lifted Planning. In *Proc. AAAI 2024*, 20078–20086.

Chen, D. Z.; Trevizan, F.; and Thiébaux, S. 2024. Return to Tradition: Learning Reliable Heuristics with Classical Machine Learning. In *Proc. ICAPS 2024*, 68–76.

Chrestien, L.; Edelkamp, S.; Komenda, A.; and Pevný, T. 2023. Optimize Planning Heuristics to Rank, not to Estimate Cost-to-Goal. In *Proc. NeurIPS 2023*.

Corrêa, A. B.; Francès, G.; Pommerening, F.; and Helmert, M. 2021. Delete-Relaxation Heuristics for Lifted Classical Planning. In *Proc. ICAPS 2021*, 94–102.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In *Proc. ICAPS 2020*, 80–89.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2022. The FF Heuristic for Lifted Classical Planning. In *Proc. AAAI 2022*, 9716–9723.

Dedieu, A.; Mazumder, R.; and Wang, H. 2022. Solving L1-regularized SVMs and Related Linear Programs: Revisiting the Effectiveness of Column and Constraint Generation. *JMLR*, 23: 1–41.

Drexler, D.; Seipp, J.; and Geffner, H. 2024. Expressing and Exploiting Subgoal Structure in Classical Planning Using Sketches. *JAIR*, 80: 171–208.

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 *Proc. ICAPS 2022*, 583–587.

Francès, G.; Corrêa, A. B.; Geissmann, C.; and Pommerening, F. 2019. Generalized Potential Heuristics for Classical Planning. In *Proc. IJCAI 2019*, 5554–5561.

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2016. Learning to Rank for Synthesizing Planning Heuristics. In *Proc. IJCAI 2016*, 3089–3095.

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.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N.; Holte, R. C.; and Schaeffer, J. 2014. Enhanced Partial Expansion A*. *JAIR*, 50: 141–187.

Hao, M.; Trevizan, F.; Thiébaux, S.; Ferber, P.; and Hoffmann, J. 2024. Guiding GBFS Through Learned Pairwise Rankings. In *Proc. IJCAI 2024*.

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.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *Proc. AAAI 2021*, 8064–8073.

Lauer, P.; Torralba, Á.; Fišer, D.; Höller, D.; Wichlacz, J.; and Hoffmann, J. 2021. Polynomial-Time in PDDL Input Size: Making the Delete Relaxation Feasible for Lifted Planning. In *Proc. IJCAI 2021*, 4119–4126.

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

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *JAIR*, 67: 129–167.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proc. ICAPS 2020*, 574–584.

Shen, W.; Trevizan, F.; Toyer, S.; Thiébaux, S.; and Xie, L. 2019. Guiding Search with Generalized Policies for Probabilistic Planning. In *Proc. SoCS 2019*.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022a. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *Proc. ICAPS 2022*, 629–637.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022b. Learning Generalized Policies without Supervision Using GNNs. In *Proc. KR 2022*, 474–483.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2023. Learning General Policies with Policy Gradient Methods. In *Proc. KR 2023*.

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 Magazine*, 1–17.

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *JAIR*, 68: 1–68.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *Proc. AAAI 2018*, 6294–6301.

Wang, R. X.; and Trevizan, F. 2025a. Code for the ICAPS-25 paper "Leveraging Action Relational Structures for Integrated Learning and Planning". https://doi.org/10.5281/zenodo.15302015.

Wang, R. X.; and Trevizan, F. 2025b. Leveraging Action Relational Structures for Integrated Learning and Planning. arXiv:2504.20318 [cs.AI].

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with Partial Expansion for Large Branching Factor Problems. In *Proc. AAAI 2000*, 923–929.