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

Continuing the Quest for Polynomial Time Heuristics in PDDL Input Size: Tractable Cases for Lifted h^{add}

Pascal Lauer[1,2], **Álvaro Torralba**[3], **Daniel Höller**[1], **Jörg Hoffmann**[1,4]

[1]Saarland Informatics Campus, Saarland University, Saarbrücken, Germany
[2]School of Computing, The Australian National University, Canberra, Australia
[3]Aalborg University, Aalborg, Denmark
[4]German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
[1]lastname@cs.uni-saarland.de [2]firstname.lastname@anu.edu.au [3]alto@cs.aau.dk

Abstract

Recent interest in solving planning tasks where full grounding is infeasible has highlighted the need to compute heuristics at a lifted level. We turn our attention to the evaluation of the h^{add} heuristic, which is an important cornerstone in many classical planning approaches, including the best performing lifted planning approach. We show that h^{add}'s grounded efficiency does not extend to lifted tasks, where the computation is EXPTIME-complete. This prompts to identify tractability islands matching practical use cases. We identify two, where a lifted computation is feasible while grounding may fail: The first constraints to acyclic action schemata and bounds predicate arity. For the second case we introduce a novel computation, operating without grounding. Assuming the extraction encounters only acyclic conditions, and h^{add} values per subgoal are bounded, it remains tractable, even if predicate and action arity is unbounded. In an empirical evaluation of the new technique, we observe complementary behavior to the existing lifted forward h^{add} evaluation. Combining both sets a new state of the art in pure-heuristic performance on the hard-to-ground benchmarks.

1 Introduction

Since the introduction of STRIPS (Fikes and Nilsson 1971) planning tasks are specified in a lifted format, where variables act as placeholders for objects in all grounded variants. Today, most planners accept PDDL (McDermott et al. 1998) as lifted input and then ground the task. Grounding enables the use of polynomial-time heuristics (Hoffmann and Nebel 2001; Helmert and Domshlak 2009; Seipp and Helmert 2018). However, grounding can come at exponential cost and has long been recognized as a theoretical bottleneck (Erol, Nau, and Subrahmanian 1995; Helmert 2009). In recent years, this bottleneck has become increasingly evident in various applications (Areces et al. 2014; Masoumi, Antoniazzi, and Soutchanski 2015; Wichlacz, Torralba, and Hoffmann 2019). This has driven focus towards creating planners that circumvent grounding to the extent possible. In particular, Corrêa et al. (2020) introduced an effective forward search by linking state expansion to conjunctive query evaluation. To replicate the success of grounded heuristics, it is necessary to not only adapt the heuristic to the lifted setting but also maintain the polynomial-time guarantee. So far, it was only observed how to: Recreate grounded heuristics exactly, without a polynomial-time guarantee (Corrêa et al. 2021, 2022). Or, apply a relaxation to ensure a polynomial-time computation, but resulting in weaker heuristics (Lauer et al. 2021). In this paper, we investigate to which extent it is possible to compute h^{add} (Bonet and Geffner 2001) in polynomial-time at the lifted level.

h^{add} is a cornerstone of classical planning computations, such as h^{FF} (Hoffmann and Nebel 2001). The lifted variant by Corrêa et al. (2021) is particularly significant in our setting. It creates the top-performing configuration when used in BFWS (Lipovetzky and Geffner 2017; Corrêa and Seipp 2022) on the hard-to-ground benchmarks. Here, h^{add} vastly dominates the overall runtime, making it the prime target for improvements. Unlike in the grounded setting, reducing the runtime of lifted h^{add} is crucial. The current lifted-forward evaluation can take minutes for a single state evaluation on hard-to-ground benchmarks. We show that this bottleneck is generally unavoidable, as computing h^{add} at a lifted level is EXPTIME-complete. Though, in some domains, the computation of Corrêa et al. (2021) is notably fast. Considering these cases, we identify a tractable subclass by bounding predicate arity and restricting to acyclic action schemata.

Recognizing this strength, we turn to the approach's main weakness: Its tie to exploring all facts. Therefore we introduce an alternative lifted h^{add} computation that avoids this bottleneck. Our approach works backwards (using *regression*). Starting from the goal, it gradually builds new conditions, represented by lifted atoms, until one is satisfied. This allows to generate only one grounded certificate to show a condition is true, and eliminates the tie to all facts. Many early approaches in planning operated on the lifted representation and used regression (McDermott 1996; Penberthy and Weld 1992; Vieille 1986; Younes and Simmons 2003). But they faced bottlenecks from partial grounding. Our approach circumvents this bottleneck by operating without grounding.

We implement our computation with optimizations to make it practical and observe complementary behavior to the forward evaluation. Our method is significantly faster in some domains, while the latter remains dominant in overall performance. We introduce a simple combination of both, leveraging their individual strengths. This advances the state-of-the-art in pure-heuristic performance on hard-to-ground benchmarks. The results indicate that other lifted

approaches utilizing h^{add} could significantly benefit from this in future work, such as the top-performer using BFWS.

2 Background

A lifted planning task is a tuple $\Pi = (\mathcal{P}, \mathcal{O}, \mathcal{A}, \mathcal{I}, \Gamma)$, where $\mathcal{P}$ is a set of predicates, $\mathcal{O}$ is a set of objects, $\mathcal{A}$ is a set of lifted actions, $\mathcal{I}$ is called initial state and Γ is called goal. A predicate $p \in \mathcal{P}$ has a fixed arity $|p|$. We assume an infinite set of variables X. The combination $p(\vec{x})$ of a predicate p with a tuple $\vec{x}$, of $|p|$ variables or objects is called lifted atom. $\mathcal{P}^X$ is the set of all lifted atoms. $X(p(\vec{x}))$ denotes the set of variables of $p(\vec{x})$. For a set of lifted atoms $L \subseteq \mathcal{P}^X$, the set of all contained variables is $X(L)$. $p(\vec{x})$ is called grounded (or fact) if $X(p(\vec{x})) = \emptyset$. We indicate this, where relevant, by $p(\vec{o})$. The set of all grounded atoms is denoted as $\mathcal{P}^{\mathcal{O}}$. A state, including $\mathcal{I}$, is a set of grounded atoms. The set of all states is S. The goal Γ is a set of grounded atoms. An action $a \in \mathcal{A}$ is a tuple $(pre(a), add(a), del(a), c(a))$, where $pre(a), add(a), del(a)$ are sets of lifted atoms. $c(a) \in \mathbb{R}_0^+$ is the action cost. $X(a) := X(pre(a) \cup add(a) \cup del(a))$ denotes the variables occuring in a. An action a is grounded if all of its atoms are grounded. A task is grounded if all of its actions are grounded.

Actions and atoms can be grounded by substituting their variables with objects. We formalize the substitution similar to McDermott (1996) by defining the possible substitutions $\mathrm{match}(\delta) := \{\theta : X(\delta) \to \mathcal{O}\}$ for the variables in some structure $\delta \in \mathcal{P}^X \cup \mathcal{A} \cup 2^{\mathcal{P}^X}$ with objects $\mathcal{O}$. $\theta(\delta)$ denotes the application of $\theta \in \mathrm{match}(\delta)$ to δ. $X(\theta)$ denotes the domain of θ. This definition extends to substitutions $\sigma : X(\delta) \to \mathcal{O} \cup X$, that are not guaranteed to ground. A grounded task is obtained by replacing the actions a by all grounded actions $\mathcal{A}^{\mathcal{O}} := \{\theta(a) \mid a \in \mathcal{A}, \theta \in \mathrm{match}(a)\}$.

A grounded action $\theta(a) \in \mathcal{A}^{\mathcal{O}}$ can be applied to a state $s \in S$ if $pre(\theta(a)) \subseteq s$ to obtain the successor $\mathrm{progr}(s, a) := (s \setminus del(\theta(a)) \cup add(\theta(a)))$. We denote $\mathrm{progr}(\mathrm{progr}(\mathrm{progr}(s, \theta_1(a_1)), ...), \theta_n(a_n))$ by $\mathrm{progr}(s, \theta_1(a_1), ..., \theta_n(a_n))$. A plan is an action sequence $\theta_1(a_1), \ldots, \theta_n(a_n)$ so that $\mathrm{progr}(\mathcal{I}, \theta_1(a_1), \ldots, \theta_n(a_n)) \supseteq \Gamma$. A task is solved by finding a plan.

A predicate $p \in \mathcal{P}$ (or atom $p(\vec{x})$) is static iff no action $a \in \mathcal{A}$ has an atom $p(\vec{y}) \in add(a) \cup del(a)$. For a set of lifted atoms $L \subseteq \mathcal{P}^X$ the set of its static atoms is $static(L)$. An action $a \in \mathcal{A}$ has distinct precondition groundings, iff for all of its groundings $\theta(a) \in \mathcal{A}^{\mathcal{O}}$ the number of its non-static preconditions remains the same after grounding, i.e., $|pre(a) \setminus static(pre(a))| = |pre(\theta(a)) \setminus static(pre(\theta(a)))|$. We assume all actions to have distinct precondition groundings. Otherwise, the value of lifted h^{add} computations, including that by Corrêa et al. (2021), may deviate.

2.1 Grounded Computation of h^{add}

h^{add} (Bonet and Geffner 2001) is the point-wise greatest function fulfilling for a state $s \in S$ and $G \subseteq \mathcal{P}^{\mathcal{O}}$:

$$h^{add}(s, G) = \begin{cases} 0 & \text{, if } G \subseteq s \\ \sum_{p(\vec{o}) \in G} h^{add}(s, \{p(\vec{o})\}) & \text{, if } |G| \neq 1 \\ \min\limits_{\theta(a) \in \mathrm{ach}(G)} c(a) + h^{add}(s, pre(\theta(a))) & \text{, o/w} \end{cases}$$

Where $\mathrm{ach}(\{p(\vec{o})\}) = \{\theta(a) \in \mathcal{A}^{\mathcal{O}} \mid p(\vec{o}) \in add(\theta(a))\}$ defines the achievers for a grounded atom $p(\vec{o}) \in \mathcal{P}^{\mathcal{O}}$.

The heuristic value of s is $h^{add}(s) := h^{add}(s, \Gamma)$. In grounded planning, h^{add} is computed by a bottom up dynamic programming approach, tagging each propositional fact $p(\vec{o}) \in \mathcal{P}^{\mathcal{O}}$ with a value converging to $h^{add}(s, \{p(\vec{o})\})$. It begins by initializing all facts $p(\vec{o}) \in s$ with value 0. All other facts are set to an initial value ∞. Iteratively, for a fact $p(\vec{o}) \in \mathcal{P}^{\mathcal{O}}$ and grounded action $a \in \mathcal{A}^{\mathcal{O}}$ that achieves the fact, i.e. $p(\vec{o}) \in add(a)$, we update the fact value to be the sum of the h^{add} values of its preconditions, plus the action cost, if that results in lower value. Performing this update in a uniform-cost manner, where only the fact with the smallest new value is updated, ensures that each fact is updated only once and ensures to run in polynomial time (w.r.t. the ground task). Finally, the heuristic value for s is determined by summing the h^{add} values of all goal facts $p(\vec{o}) \in \Gamma$.

As the computation ignores delete effects, it is called delete-relaxed. The delete relaxation Π^+ of task Π is obtained by removing the delete lists of all actions.

2.2 Satisfiability and Query Evaluation

The set of substitutions $\mathrm{match}(\delta) := \{\theta : X(\delta) \to \mathcal{O}\}$ for structures $\delta \in \mathcal{P}^X \cup \mathcal{A} \cup 2^{\mathcal{P}^X}$ grows exponentially in the size of δ. So, generating the entire set is asymptotically as expensive as grounding the task. Though, in most cases only a small subset of variable replacement functions is relevant. These functions are restricted by some condition expressed as a set of lifted atoms $L \subseteq \mathcal{P}^X$ and evaluated over a given state $s \in S$. Formally we set $\mathrm{match}(s, L) := \{\theta \in \mathrm{match}(L) \mid \theta(L) \subseteq s\}$. A good example for such a restriction is given in Corrêa et al. (2020). They use $\mathrm{match}(s, pre(a))$ to determine applicable grounded actions for some lifted $a \in \mathcal{A}$ in a state $s \in S$. Here we link this evaluation to terminology and well-known theoretical results from database theory, which are commonly found in introductory books, including Abiteboul, Hull, and Vianu (1995). To provide context, a conjunctive query in these texts is essentially a set of lifted atoms $L \subseteq \mathcal{P}^X$, our states serve as database, tables are a collection of variable replacement functions and $\mathrm{match}(s, L)$ is the answer to the query L. We say that some state s satisfies L, denoted by $s \vDash L$ iff $\mathrm{match}(s, L) \neq \emptyset$. Checking satisfiability $s \vDash L$ is NP-complete (Chandra and Merlin 1977) indicating the existence of a polynomial-sized certificate $\theta \in \mathrm{match}(s, L)$ iff $\mathrm{match}(s, L) \neq \emptyset$. Moreover, there exist many practical criteria for L, that if fulfilled, ensure that the computation of some $\theta \in \mathrm{match}(s, L)$ runs in polynomial time (Flum, Frick, and Grohe 2002; Grohe, Schwentick, and Segoufin 2001). Still, the enumeration of all results $\mathrm{match}(s, L)$ can be exponential in $|X(L)|$, even in these tractable cases. This advantage of easy satisfaction over an intractable enumeration is a key motivation for the h^{add} computation we introduce in Section 4.

A commonly exploited tractable case is acyclicity, e.g. see Beeri et al. (1981). A set of lifted atoms $L \subseteq \mathcal{P}^X$ is acyclic iff there exists a tree with nodes L so that for all $p_1(\vec{x}_1), p_2(\vec{x}_2) \in L, x \in X(p_1(\vec{x}_1)) \cap X(p_2(\vec{x}_2))$, x occurs

in every node on the shortest path from $p_1(\vec{x}_1)$ to $p_2(\vec{x}_2)$. In this case it is possible to check satisfiability in polynomial time via the GYO algorithm (Graham 1979; Yu and Ozsoyoglu 1979). We call a planning task acyclic if all $pre(a)$ for $a \in \mathcal{A}$ are acyclic. The successor generation of Corrêa et al. (2020) capitalized of the fact that acyclic planning tasks are common in the International Planning Competition (IPC) benchmark set by using Yannakakis algorithm (Yannakakis 1981), which is polynomially bounded in both input and output. To limit the output, we define the projection over variables $Y \subseteq X(\theta)$ as $\pi_Y(\theta) = \{(x \mapsto o) \in \theta \mid x \in Y\}$. If Y is restricted in size, then the output is guaranteed to be bounded w.r.t. the lifted task representation and Yannakakis algorithm can evaluate $\pi_Y(\mathrm{match}(s, L))$ in polynomial time. If we associate a weight $w : s \to \mathbb{R}$ with every element of a state $s \in S$, e.g. be the h^{add} cost of each atom in the state, Yannakakis algorithm can be adapted to compute the cheapest cost $\min_{\theta \in \mathrm{match}(s,L)} \sum_{p(\vec{x}) \in L} w(\theta(p(\vec{x})))$ in polynomial time.

$\mathcal{P}^{X+}$ is the set of all multisets over $\mathcal{P}^X$. All introduced definitions, including algorithms for satisfiability and query evaluation, transfer canonically from sets to multisets as the number of elements in multisets is irrelevant in this context (but will be relevant for our algorithm).

3 The Complexity of Computing Lifted h^{add}

In this section, we prove that computing h^{add} on lifted planning tasks is EXPTIME-complete, identify a task restriction that allows tractable forward computation, and discuss when and why such forward evaluations, including Corrêa et al.'s (2021) approach, encounter significant limitations. Due to space constraints, we present proof sketches throughout the whole paper. Full proofs are provided in the Appendix (Lauer et al. 2025).

Theorem 1. *Computing h^{add} on lifted planning tasks is EXPTIME-complete.*

Proof sketch. Hardness follows by reduction from delete-relaxed plan existence which is EXPTIME-complete (Erol, Nau, and Subrahmanian 1995). (A delete-relaxed plan exists for $s \in S$ iff $h^{add}(s) \neq \infty$.) Membership is proven by grounding the task in time exponential in the lifted representation to compute grounded h^{add}. □

We now identify a task restriction that makes the computation tractable using a forward fix-point approach. We adapt the idea of the grounded computation to generate the reachable facts on demand from the lifted action representation. In particular, we use Yannakakis algorithm under acyclicity. Additionally fixing predicate arity, to limit the number of facts, makes the computation tractable.

Theorem 2. *h^{add} can be computed in polytime on acyclic lifted planning tasks with predicate arity at most $k \in \mathbb{N}$.*

Proof sketch. We adapt the grounded h^{add} computation from Section 2.1 to consider grounded facts, but not grounded actions. For each action schema, we can generate the cheapest sum of h^{add} values for any precondition of a grounded operator of the schema, by Yannakakis algorithm. The restriction to acyclic preconditions allows a single evaluation to run in polynomial time. The limit on predicate arity constrains the number of facts, resulting in polynomially many iterations and an overall polynomial runtime. □

make_sandwich($x_{c1} \dots x_{cn}$ – content):
pre : {inKitchen(x_{c1}), …, inKitchen(x_{cn})}
add : {sandwich($x_{c1}, \dots, x_{cn}$)}
del : {inKitchen(x_{c1}), …, inKitchen(x_{cn})}

serve_sandwich(x_{ch} – child $x_{c1} \dots x_{cn}$ – content):
pre : {likes(x_{ch}, x_{c1}), …, likes(x_{ch}, x_{cn}), sandwich($x_{c1}, \dots, x_{cn}$)}
add : {served(x_{ch})}
del : {sandwich($x_{c1}, \dots, x_{cn}$)}

Figure 1: Actions for the Childsnack running example.

Corrêa et al. (2021) use a similar approach to compute h^{add}. To handle actions with large arity they transform the task, decomposing actions according to Helmert (2009) into smaller ones, with fewer parameters. In the transformed task, each fact produced by the forward evaluation mirrors an intermediate result of a query evaluation. If the arity of intermediate results is bounded, the computation remains tractable; just as query evaluations like GYO and Yannakakis achieve tractability by bounding intermediate results if the query is acyclic. We conducted an analysis on hard-to-ground benchmarks to determine cases where using Yannakakis algorithm would reduce this bound and found that, even though they are not necessarily equal in the worst-case, they are practically indistinguishable on these benchmarks. Thus, our proof does not only mark a new tractability fragment, but also clarifies where the forward evaluation by Corrêa et al. (2021) works well.

Note that acyclic conditions and bounded predicate arity need to be combined. If we would just bound predicate arity, delete-relaxed plan existence remains NP-hard since conditions could be cyclic (Lauer et al. 2021). Acyclic conditions alone do not guarantee tractability, as any planning task can be made acyclic by increasing predicate arity.

Proposition 3. *For any lifted planning task Π, there is an acyclic planning task Π' with size at most $O(|\Pi|)$ such that any plan in Π can be rewritten to a plan in Π' and vice versa.*

Proof sketch. Obtain Π' from Π by making the precondition of each action a with $X(a) = \{x_1, ..., x_n\}$ acyclic by adding an atom $p_a(x_1, ..., x_n)$. Further, add new actions u_{p_a} of cost 0 that make $p_a(x_1, ..., x_n)$ achievable by having one add element $p_a(x_1, ..., x_n)$, but no precondition. Thus a plan in Π can be converted to a plan in Π' by adding $u_{p_a}(o_1, ..., o_n)$ before each $a(o_1, ..., o_n)$ in the plan. The plan conversion from Π' to Π works by dropping all u_{p_a} actions. □

The lifted search by Corrêa et al. (2020) already assumes acyclic preconditions, which makes this requirement natural. Though, the restriction to bounded predicate arity does not always match reality. We will examine this

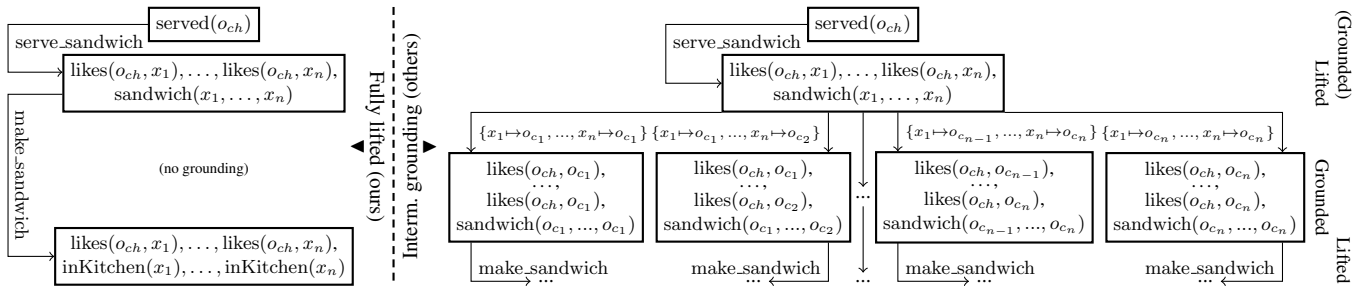

Figure 2: The lext regression graph (left). Related approaches with exponential node explosion in one intermediate grounding step (right).

on a simplified version of Childsnack from the hard-to-ground benchmark set, as shown in Figure 1. The task is to make sandwiches with multiple contents $o_{c_1}, \ldots, o_{c_n}$ connected by the sandwich predicate and serve them, denoted by the served predicate, to children o_{ch} based on their preferences denoted by likes. In order to serve a child with serve_sandwich, the sandwich has to be created using make_sandwich. The available contents are denoted by the predicate inKitchen.

A lifted forward exploration would first create all possible combinations for sandwich atoms sandwich$(o_{c_1}, \ldots, o_{c_n})$. This grows exponentially in n, making the forward evaluation infeasible. The problem is closely linked to the bottleneck a grounder encounters when grounding all actions. Proposition 3 makes this link explicit by using predicates of same arity as actions. Here the challenge is to identify a subset of relevant actions rather than all applicable ones. However, all h^{add} evaluations thus far are tied to exploring all facts. In the remainder of the paper, we will develop an evaluation without this drawback.

4 A Lifted Regressive Formulation for h^{add}

In this section, we provide an alternative h^{add} formulation. As it builds the foundation for our Lifted Regressive computation, we call it h^{LRadd}. Proceeding backwards, i.e. using regression, allows to omit the intermediate grounding step forward approaches are linked to. This allows h^{LRadd} to operate independent of the number of delete-relaxed reachable atoms. To define h^{LRadd}, we introduce the lext (lifted extension) function, representing a lifted regression tree where nodes are multisets of lifted atoms $L \in \mathcal{P}^{X+}$. Each node represents a sufficient condition for delete-relaxed plan existence. Starting from the goal, lext recursively defines new conditions by replacing exactly one atom with the lifted precondition of an action that can achieve it. This ensures that if $s \vDash L$ for any node, a (relaxed) plan exists. When each node is linked to the total cost of actions along its creation path (lext path), h^{LRadd} represents the lowest cost for a node to be satisfied. We prove in Section 5 that $h^{LRadd}(s) = h^{add}(s)$ for all $s \in S$.

Figure 2 (to the left) shows the lext tree for our running example, with one goal served(o_{ch}). The only action that may achieve the goal served(o_{ch}) is serve_sandwich. We replace the atom served(o_{ch}) with the precondition of serve_sandwich, replacing the first parameter to match served(o_{ch}). If the node is satisfied, then the heuristic value is 1. Otherwise, we need to explore the tree by considering all options of achieving an atom in the node by some action. In this case, the only predicate appearing in the add list of an action and the node is sandwich. The corresponding atom sandwich$(x_1, \ldots, x_n)$ is subsequently replaced by make_sandwich in the next step.

The tree to the right of Figure 2 illustrates the exponential blow up that would occur in a single step, when integrating intermediate grounding step into the backwards approach. We will prove in Theorem 5 of section 5, that the fully lifted tree to the left implicitly represents the complete grounded tree to the right. The lifted representation allows to be more compact by shifting complexity from search to node evaluation. In many planning tasks, the additional evaluation introduces at most polynomial overhead, as it aligns with tractable evaluation cases, like in our running example where all nodes are acyclic. The grounded case makes it more obvious that this closely aligns with a delete-relaxed plan extraction. A delete-relaxed plan corresponds exactly to the grounded actions used to replace atoms along the path to a satisfied node, compare Suda (2016).

Before defining the regression graph, we must address the technicality of remapping variables of an action $a \in \mathcal{A}$, so that the $add(a)$ variables match the replaced $p(\vec{x}) \in L$ of the condition $L \in \mathcal{P}^{X+}$ to extend (C1). Any other $pre(a)$ variables should be each replaced by a unique new variable that is not already part of $L \in \mathcal{P}^{X+}$, to avoid additionally restrictions among variables (C2,C3). For convenience, we assume $|add(a)| = 1$ for $a \in \mathcal{A}$. This is achievable by a simple task transformation preserving h^{add} (Corrêa et al. 2021).

Definition 1. *Let $p(\vec{x}) \in L \in \mathcal{P}^{X+}$ and $a \in \mathcal{A}$. A mapping $\sigma : X(a) \to X \cup \mathcal{O}$ is valid for a, $p(\vec{x})$ and L if:*

(C1) $p(\vec{x}) \in add(\sigma(a))$
(C2) $X(\sigma(a)) \cap X(L) = X(p(\vec{x}))$
(C3) $\sigma_{|X(a) \setminus X(add(a))}$ is injective

The set VarRemap$(L, p(\vec{x}), a)$ *denotes all valid variable mappings for $p(\vec{x}) \in L, a \in \mathcal{A}$.*

To define the regression graph via lext for a set $L \in \mathcal{P}^{X+}$, we define concrete transitions lext$(L, p(\vec{x}), a)$ if $a \in \mathcal{A}$ achieves some $p(\vec{x}) \in L$ and their collection lext(L). A path in the graph starting at L is called an lext-path from L.

Definition 2. $\text{lext}(L, p(\vec{x}), a)$*, i.e. the lifted regressive extension for* $p(\vec{x}) \in L \in \mathcal{P}^{X+}$ *over* $a \in \mathcal{A}$*, is defined as:*

$$\text{lext}(L, p(\vec{x}), a) = \begin{cases} \bot \text{ , if } \nexists \sigma \in \text{VarRemap}(L, p(\vec{x}), a) \\ L \setminus \{p(\vec{x})\} \cup \sigma(pre(a)) \quad \text{, o/w} \end{cases}$$

We extend this to obtain all applications $\neq \bot$ *as* $\text{lext}(L) := \{(a, L') \mid \bot \neq \text{lext}(L, p(\vec{x}), a) = L', a \in \mathcal{A}, p(\vec{x}) \in L\}$.

Note that the choice $\sigma \in \text{VarRemap}(L, p(\vec{x}), a)$ is not unique. In particular, names of variables that were not part of L before are undetermined. We fix the choice to the smallest x_i for $i \in \mathbb{N}$ that are not part of L, like in the running example. Any other choice creates an isomorphic structure that only differs in the names chosen for new variables. We define h^{LRadd} as the cheapest cost to a satisified node L when adding up the action cost along the path to L.

Definition 3. *The value of* h^{LRadd} *is defined as the pointwise greatest function satisfying the following equation:*

$$h^{LRadd}(s, L) = \begin{cases} 0 & \text{, if } s \vDash L \\ \min\limits_{(a,L') \in \text{lext}(L)} c(a) + h^{LRadd}(s, L') & \text{, o/w} \end{cases}$$

Furthermore, we set: $h^{LRadd}(s) = h^{LRadd}(s, \Gamma)$.

5 Equality to h^{add}

Here we will prove that h^{add} and h^{LRadd} produce the same heuristic values. We distinguish between the value $h^{add}(s)$ being infinity or not for a fixed state $s \in S$. We start with $h^{add}(s) = \infty$.

Proposition 4. $\forall s \in S : h^{add}(s) = \infty \Rightarrow h^{LRadd}(s) = \infty$

Proof sketch. It is known that $h^{add}(s) \neq \infty$ if and only if there is a delete-relaxed plan for s. If $h^{LRadd}(s) \neq \infty$, then there is a sequence of $L_0, \ldots, L_n \in \mathcal{P}^{X+}$ such that $L_n = \Gamma$, $L_{i-1} = \text{lext}(L_i, p(\vec{x})_i, a_i)$ (using σ_i), and $s \vDash L_0$ (with $\theta_0 \in \text{match}(s, L_0)$). Then, the sequence $\theta_0(\sigma_1(a_1)), \ldots, \theta_{n-1}(\sigma_{n-1}(a_n))$ with $\theta_i \supseteq \theta_{i-1|X(\theta_i)}$ for $i \in \mathbb{N}^+$ is a delete-relaxed plan for s, contradicting $h^{add}(s) = \infty$. □

We now consider the case $h^{add}(s) \neq \infty$. Lemma 5 shows that the value of h^{LRadd} does not change under intermediate grounding as illustrated in Figure 2.

Lemma 5. $\forall s \in S,\ L \in \mathcal{P}^{X+}$*:*

$$h^{LRadd}(s, L) = \min_{\theta \in \text{match}(L)} h^{LRadd}(s, \theta(L))$$

Proof sketch. It holds that $h^{LRadd}(s, L) \leq \min_{\theta \in \text{match}(L)} h^{LRadd}(s, \theta(L))$, as any $\theta(L)$ is just a more constrained version of L. To show the other direction, consider any lext-path $L_0, \ldots, L_n$ where $L_n = \Gamma$, $L_{i-1} = \text{lext}(L_i, p(\vec{x})_i, a_i)$ with $\sum_i c(a_i) = h^{LRadd}(s, L)$ and $s \vDash L_0$. Let θ^* be a variable replacement under which $s \vDash L_0$. Then, $h^{LRadd}(s, L) = h^{LRadd}(s, \theta^*(L)) \geq \min_{\theta \in \text{match}(L)} h^{LRadd}(s, \theta(L))$. □

Now we prove that for grounded sets we can split up the computation like in the h^{add} definition. Combining this result with Lemma 5 proves our main claim.

Lemma 6. $\forall s \in S,\ G \in \mathcal{P}^{\mathcal{O}+}$*:*

$$h^{LRadd}(s, G) = \textstyle\sum_{p(\vec{o}) \in G} h^{LRadd}(s, \{p(\vec{o})\})$$

Proof sketch. For convenience, we denote single atom sets $\{p_1(\vec{o}_1)\}$, ..., $\{p_n(\vec{o}_n)\}$ composing $G = \{p_1(\vec{o}_1)\} \cup \cdots \cup \{p_n(\vec{o}_n)\}$ by $G_1, ..., G_n$, rewriting the equation to:

$$h^{LRadd}(s, G) = \sum\nolimits_{i=1}^{n} h^{LRadd}(s, G_i)$$

The proof relies on the fact that sets without shared variables, and their replacements, can be independently split and recombined. There are no shared variables between the sets $G_1, \ldots, G_n$ as each set contains exactly one grounded atom, so no variables at all. We begin by making three observations for a generic set of lifted atoms $L = L_A \cup L_B$ with disjoint variables, i.e. $X(L_A) \cap X(L_B) = \emptyset$, which generalize to $G_1, \ldots, G_n$ and their replacements.

(Obs1): If two set of lifted atoms do not share any variables, then satisfiability can be evaluated independently. In particular, it holds that $s \vDash L \Leftrightarrow s \vDash L_A \wedge s \vDash L_B$.

(Obs2): If two atoms do not share variables, their replacements never will. The reason is that variables in the replacement are either those already used in the predicate being replaced or completely new variables. From this it follows that replacements of L_A and L_B will never share variables.

(Obs3): In a set of lifted atoms variable names are arbitrary. So we can always apply any bijective variable substitution to a new set of variables. For example, $p(x_1, x_2), q(x_2)$ is equivalent to $p(x_2, x_3), q(x_3)$, i.e. they are satisfied in the same states.

To prove $\geq$, we split any lext-path from G into subpaths, each corresponding to the (possibly recursive) replacement of one G_i for $i \in \{1, \ldots, n\}$. By (Obs2), the result of each such path introduces a unique set of variables disjoint from the others. Hence, by (Obs1), satisfiability holds for each path independently. For each $i \in \{1, \ldots, n\}$, the cost c_i of the subpath originating from L_i provides an upper bound on $h^{LRadd}(s, G_i)$, as the path ends in a satisfied set of atoms. Thus the sum over all subpaths $h^{LRadd}(s, G) = \sum_{i=1}^{n} c_i \geq \sum_{i=1}^{n} h^{LRadd}(s, G_i)$.

To prove $\leq$, we consider a cheapest lext-paths for $G_1, \ldots, G_n$ with costs $c_1, ..., c_n$ individually. By (Obs3), we can apply variable substitutions to align these paths into a single lext-path from G. By (Obs1), the combined lext-path leads to a satisfied set of atoms and thus provides an upperbound on $h^{LRadd}(s, G)$, i.e. $h^{LRadd}(s, G) \leq \sum_{i=1}^{n} c_i = \sum_{i=1}^{n} h^{LRadd}(s, G_i)$.

This concludes the proof. Note that in the full technical proof in the Appendix, this argument is applied in the proof of Theorem 11, which states a more general version of Lemma 6 for arbitrary sets of lifted atoms with disjoint variables. By proving the general case once, we avoid a duplication of the technical proof in the Appendix. □

Theorem 7. $\forall s \in S : h^{add}(s) = h^{LRadd}(s)$

Proof sketch. Proposition 4 covers the case $h^{add}(s) = \infty$. For $h^{add}(s) \neq \infty, L \in \mathcal{P}^{X+}$ we prove $h^{LRadd}(s, L) = \min_{\theta \in \text{match}(L)} h^{add}(s, \theta(L))$ which proves the claim.

Proof via induction over the minimal recursion depth of any $h^{add}(s, \theta(L))$ computation. For the base case (depth 0) we know that there is a $\theta \in \text{match}(L)$ so that $\theta(L) \subseteq s$ and thus $h^{LRadd}(s, L) = 0 = \min_{\theta \in \text{match}(L)} h^{add}(s, \theta(L))$.

For the induction step we distinguish between $|L| = 1$ and $|L| > 1$. For $|L| = 1$, i.e. just one atom, it is easy to observe (Obs.) that (1) grounding the atom to determine the precondition of the achiever action and (2) determining the lifted precondition via lext to ground afterwards both return the grounded precondition of an action that can achieve the atom. [1] Thus:

$$\begin{aligned}
&\min_{\theta \in \text{match}(L)} h^{add}(s, \theta(L)) && [\text{ Def. } h^{add} \text{ }]\\
&= \min_{\theta \in \text{match}(L)} \min_{\theta'(a) \in \text{ach}(\theta(L))} c(a) + h^{add}(s, pre(\theta'(a))) && [\text{ Obs. }]\\
&= \min_{(a, pre(a)) \in \text{lext}(L)} \min_{\theta' \in \text{match}(a)} c(a) + h^{add}(s, pre(\theta'(a)))\\
&= \min_{(a, pre(a)) \in \text{lext}(L)} c(a) + h^{LRadd}(s, pre(a)) && [\text{ I.H. }]\\
&= h^{LRadd}(s, L) && [\text{ Def. } h^{LRadd} \text{ }]
\end{aligned}$$

For $|L| > 1$:

$$\begin{aligned}
&\min_{\theta \in \text{match}(L)} h^{add}(s, \theta(L))\\
&= \min_{\theta \in \text{match}(L)} \sum_{p(\vec{x}) \in L} h^{add}(s, \theta(\{p(\vec{x})\})) && [\text{ Def. } h^{add} \text{ }]\\
&= \min_{\theta \in \text{match}(L)} \sum_{p(\vec{x}) \in L} \min_{\theta' \in \text{match}(\theta(\{p(\vec{x})\}))} h^{add}(s, \theta'(\theta(\{p(\vec{x})\})))\\
&= \min_{\theta \in \text{match}(L)} \sum_{p(\vec{x}) \in L} h^{LRadd}(s, \theta(\{p(\vec{x})\})) && [\text{ I.H. }]\\
&= \min_{\theta \in \text{match}(L)} h^{LRadd}(s, \theta(L)) = h^{LRadd}(s, L) && [\text{ Lemma 5,6 }]
\end{aligned}$$

□

6 Computation, Runtime and Complexity

To compute h^{LRadd} on some $s \in S$, we find one cheapest satisfied node in the lext regression graph using a uniform-cost search. Upon expansion of each node $L \in \mathcal{P}^{X+}$, we check whether its query is satisfied in s using a standard database evaluation with a GYO-inspired order. If $s \vDash L$, the algorithm stops returning the node's cost. Or, if the state is a dead-end, we return ∞ whenever the search space has been exhausted or if the search depth exceeds an upper bound on the h^{add} value. We improve upon this naïve search in Section 7.

In the following, we identify a new fragment of lifted planning tasks on which computing h^{add} with our method is tractable, and connect this to existing benchmarks. The runtime of h^{LRadd} is bounded by the number of explored nodes times the runtime for the most expensive satisfiability check. To determine a bound on explored nodes while computing $h^{LRadd}(s)$, we define the set of lext nodes, E_s.

Definition 4. *Given $s \in S$ and $i \in \mathbb{N}_0$, we define the exploration layer $E_{s,i}$ of* lext *for $i = 0$ as $E_{s,0} := \{\Gamma\}$ and for $i \geq 1$ as $E_{s,i+1} := \{L' \mid (a, L') \in \text{lext}(L), L \in E_{s,i}\}$. And set $E_s := \bigcup_{i=0}^{h^{add}(s)} E_{s,i}$.*

[1] Under assumption of distinct precondition groundings.

In this analysis, we assume action costs are at least 1. This restriction aids in understanding the runtime by connecting it to h^{add} values. Specifically, the set E_s serves as a clear over-approximation of the lext-nodes explored by our method, so we can bound the number of explored nodes by bounding $|E_s|$. We omit the detailed proof here, referring to the appendix. The key observation is that the depth of our exploration corresponds to $h^{add}(s)$ and the branching factor does not scale exponentially in the size of the planning task.

Proposition 8. *Let $s \in S$. An upper-bound on $|E_s|$ is:*

$$O((|\Gamma| \cdot |\mathcal{A}| \cdot \max_{a \in \mathcal{A}} |pre(a)| \cdot \max_{a \in \mathcal{A}} |add(a)| \cdot h^{add}(s))^{h^{add}(s)})$$

At the end of the section we will observe that there are many interesting cases where the size of a lifted planning task (e.g., number of objects) and h^{add} are not inherently correlated, making the identified bound highly complementary to the one identified for the forward evaluation.

Similar to the forward evaluation, we will now formulate a tractability theorem relating to our backward evaluation. We apply the same idea of using conjunctive query evaluation only under acyclicity. Note that an acyclic task does not necessarily imply that the evaluation encounters only acyclic nodes, and vice versa. To provide a tighter bound, suitable to fit practical planning tasks, we will establish a limit for each value $h^{add}(s, \{p(\vec{o})\})$ for all subgoals $p(\vec{o}) \in \Gamma$.

Theorem 9. *h^{add} can be computed in polynomial time on lifted planning tasks with non-zero action cost where $h^{add}(s, \{p(\vec{o})\}) \leq k \in \mathbb{N}$ for all $p(\vec{o}) \in \Gamma$ and all $L \in E_s$ are acyclic.*

Proof. We can compute $h^{add}(s, \{p(\vec{o})\})$ separately for all $p(\vec{o}) \in \Gamma$. As h^{LRadd} runs in polynomial time w.r.t. the lifted planning task representation under these conditions (according to the bound on the number of nodes from Proposition 8, and since each set of lifted atoms can be checked for satisfiability in polynomial time if it is acyclic), this yields a polynomial time computation. □

Ridder and Fox (2014) and Lauer et al. (2021) used tasks with short plans, that can have very large groundings, as a key motivation for lifted planning. As short plans imply a low h^{add} value, bounding h^{add} like in Theorem 9 aligns naturally with these tasks. A prime example of this is Organic Synthesis (Masoumi, Antoniazzi, and Soutchanski 2015), where plan lengths are consistently short, often just 1-digit. Despite this, grounded planners struggle to solve the tasks, highlighting the difficulty even if plans are short.

On top of that, bounding the value per subgoal, instead of overall, allows Theorem 9 to be applicable even with scaling plan length. In the running example the task (and plan size) are scaled up by increasing the number of children to be served. The h^{add} bound remains constant per subgoal, of a child to be served, ensuring the polynomial-time guarantee. Another example are Blocksworld tasks, where all blocks start on the table and the goal is to stack towers of two. Each stacked block corresponds to a subgoal. By bounding the tower size, we bound h^{add} per subgoal, independent of how many towers need to be stacked.

In Blocksworld, predicate arity is bounded and actions are acylic. This also grants the forward evaluation a polynomial time guarantee. But in practice, this is often not enough. In the example, the forward evaluation generates all possible stack combinations, which is much harder than simply answering whether two blocks can be stacked on top of each other. As the number of blocks increases, the forward evaluation becomes significantly bottlenecked. A similar example are Logistics tasks where each package can be delivered in a constant number of steps c, regardless of the number of packages, trucks, or cities. The backward evaluation determines if any vehicle(s) can complete the delivery steps. Again, this is cheaper than navigating all vehicles through various locations for deliveries, which is what the forward method would do. Another advantage can be observed in the Gripper domain when the robot only needs to move some balls. This allows to check the specified goal only, without moving all balls, as the forward evaluation would.

On the other hand, the backward evaluation may also face limitations in other tasks, even when the polynomial time guarantee applies. E.g. in Organic Synthesis, which has many actions and preconditions, leading to a super high base factor in the bound of Proposition 8. To at least aim to decrease that a bit we will focus on how to further improve our computations in practice in the next section.

7 Making the Computation Practicable

This section introduces two optimizations to improve the practicality of our computation by restricting the exploration of the lext graph while maintaining the h^{add} value. To understand the adressed bottleneck, consider the running example, with a different goal $\{\mathrm{served}(o_{ch1}), \mathrm{served}(o_{ch2})\}$. Initially, both goal elements can be replaced, so the branching factor is 2 instead of 1. In the subsequent step, the replacement either selects the unreplaced goal $\mathrm{served}(o_{chi})$ or the obtained precondition for $\mathrm{served}(o_{chj})$. This results in a cross-product over the expansions for a single goal, causing exponential growth in the size of the goal. Both optimizations independently address and eliminate the exponential blow-up in this example in complementary ways.

7.1 Limiting the achiever selection

Our first optimization restricts the branching factor in the regression graph exploration. The following Theorem motivates to explore a subset of transitions of lext by considering only the replacements for a subset $L_S \subseteq L$ if $s \not\models L_S$ in the current $s \in S$ instead of all replacements for $L \in \mathcal{P}^{X+}$.

Theorem 10. *Let $s \in S$, $L \in \mathcal{P}^{X+}$, $L_S \subseteq L$ and $\mathrm{lext}_S := \{\mathrm{lext}(L, p(\vec{x}), a) \in \mathrm{lext}(L) \mid p(\vec{x}) \in L_S\}$. If $s \not\models L_S$, then:*

$$h^{LRadd}(s, L) = \min_{(a,L') \in \mathrm{lext}_S} c(a) + h^{LRadd}(s, L')$$

Proof sketch. This is a more general version of Lemma 6. To satisfy L, a replacement of some $p(\vec{x}) \in L_S$ must occur on the path from L to the satisfied set L'. Shifting this replacement backward or forward in the replacement order of lext does not alter the content, up to rewriting, nor the cost of L, allowing to enforce the replacement immediately. □

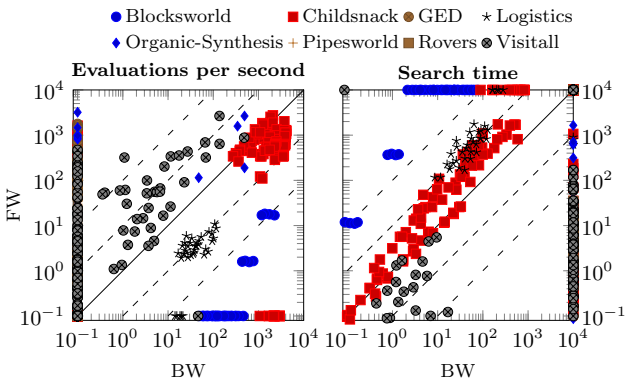

Figure 3: Comparison of our approach (BW) and the forward lifted computation (FW) by Corrêa et al. (2021) in terms of evaluations per second and search time. Search time is measured up to the last common f-layer. Data points in the extremes represent cases where one of the methods failed to compute the heuristic value for the initial state.

Thus we can simplify our exploration of lext by excluding transitions outside lext_S and still guarantee equality to h^{add}. In the modified example, if a set $L \in \mathcal{P}^{X+}$ is unsatisfied in $s \in S$, it contains either $\mathrm{sandwich}(x_1, \ldots, x_n), \mathrm{likes}(x_{ch}, x_1), \ldots, \mathrm{likes}(x_{ch}, x_n)$ or $\mathrm{served}(o_{chi})$ that is unsatisfied. These sets represent L_S where only one atom can be replaced, allowing us to explore just one node per step.

Conjunctive query evaluations, like GYO or Yannakakis, produce L_S as byproduct when L is unsatisfiable. As atoms are combined incrementally, we take L_S as those combined until proving unsatisfiability. In our example, L_S always has a single lext-successor, avoiding the exponential explosion.

7.2 Independent subset partitioning

As argued in Theorem 9, we can compute h^{add} separately for all goal atoms to avoid an exponential enumeration over all parts. This idea can be generalized to partition sets whenever they can be split into parts with disjoint parameters.

Theorem 11. *Let $L_1, L_2 \in \mathcal{P}^{X+}$, $L = L_1 \cup L_2$ and $X(L_1) \cap X(L_2) = \emptyset$, then $h^{LRadd}(s, L_1) + h^{LRadd}(s, L_2) = h^{LRadd}(s, L)$.*

Proof sketch. Note that $s \models L \Leftrightarrow s \models L_1 \wedge s \models L_2$, as there is no overlap in the variables of L_1 and L_2. Thus, by Thm. 10, one can first enforce all replacements of L_1 until satisfied, and then replace L_2 until satisfied, without changing the h^{LRadd} cost. □

This allows us to explore the smaller regression graphs underlying L_1 and L_2 independently and so implicitly enumerate all combinations, instead exploring the whole graph for L. As this eliminates explicit enumeration over all goals, it avoids the exponential enumeration in our example.

8 Experiments

To check whether the theoretical benefits of h^{LRadd} transfer to practice, we implemented it within the Powerlifted Planner (Corrêa et al. 2020), including all optimizations described in section 7. We coupled our heuristic and the heuristics used for comparison with Greedy-Best-First Search (GBFS) to evaluate its performance on the benchmark set introduced by Lauer et al. (2021). We selected the variant based on Yannakakis algorithm as successor generator. The experiments were run on a cluster of machines with Intel Xeon E5-2650 CPUs with a clock speed of 2.30GHz using the Lab framework (Seipp et al. 2017). Time and memory limits were set to 30 minutes and 4GB respectively for all runs. We verified correctness of our implementation by confirming that expansions and initial heuristic values match the lifted forward evaluation. Source code and a detailed experimental report are available on Zenodo (Lauer et al. 2025).

Figure 3 compares the number of evaluations per second and search time of our approach (BW) and the lifted forward evaluation (FW) by Corrêa et al. (2021). In order to compare the performance also on tasks that are not entirely solved, we compare the search time until the last common f-layer. As both approaches compute the same heuristic values, so the comparison is always up to expanding the exact same set of states. The evaluation presents the complementary picture we expected to find. FW and BW, show strenghts on different domains. Our approach (BW) excels at domains were the h^{add} values are small as explained in Section 6: Blocksworld, Childsnack, and Logistics. The performance in Logistics and Blocksworld is particularly impressive, with evaluation rates differing by one and two orders of magnitude respectively. The Blocksworld data points further reinforce our theoretical analysis by forming three distinct plateaus with clear gaps along the y-axis. Each gap reflects an increased difficulty level associated with adding more blocks to the problems. FW struggles to generate all possible stacking combinations. BW efficiently avoids this challenge. On the other hand, we observe a significantly decreased rate in the remaining five domains, with BW failing to produce an initial heuristic value in three domains. This aligns with our earlier analysis, as these domains exhibit longer delete-relaxed plans, and comparably fewer delete-relaxed reachable facts.

While the performance drop is not ideal, overall the results point to h^{LRadd} offering meaningful support when the lifted forward approach is less effective. To benefit from both worlds, we introduce a simple combination (COMB) that automatically picks which method to use in each instance. COMB uses both FW and BW (with optimizations enabled) for the first 10 evaluations, and times out the slower variant. The search proceeds using only the approach that was faster during those evaluations. Table 1 presents a coverage table comparing all h^{add} variants, with an ablation analysis of the optimizations of our approach (BW): limiting the achiever selection (L) from Section 7.1 and independent subset partitioning (I) from Section 7.2. As a baseline, we use the lifted forward evaluation by Corrêa et al. (2021) (FW). On top, we included a comparison to the h^{FF} variant by Corrêa et al. (2022) to evaluate the overall performance w.r.t. pure heuristic guidance. Like Höller and Behnke (2022), we normalize coverage, i.e. consider the percentages of tasks solved per domain. This reduces the influence of the huge disparity of instances per domain.

	h^{add}						h^{FF}
	Backward (BW)				FW	COMB	
	—	L	I	L + I			
Blocksworld (40)	0.0	2.5	5.0	**7.5**	2.5	**7.5**	5.0
Childsnack(144)	0.0	7.6	20.8	**24.3**	23.6	22.9	19.44
GED (312)	0.0	0.0	0.0	0.0	**43.3**	42.6	15.38
Logistics (40)	10.0	20.0	10.0	**90.0**	17.5	87.5	15.0
Org.-Synthesis (56)	0.0	0.0	5.4	7.1	80.4	80.4	**82.14**
Pipesworld (50)	0.0	0.0	0.0	0.0	40.0	40.0	**44.0**
Rovers (40)	0.0	0.0	0.0	0.0	27.5	27.5	**72.5**
Visitall (180)	7.8	10.0	17.8	20.6	**65.0**	64.4	57.22
Sum orig. (862)	18	38	71	115	370	**396**	284
Sum (862)	17.8	40.1	59.0	149.5	299.7	**372.9**	310.69

Table 1: Normalized coverage (percentage of solved instances) of different h^{add} computations: BW with each optimization enabled/disabled, FW, and their combination (COMB). Best h^{add} score has gray background. Overall best performer, compared to lifted h^{FF} (Corrêa et al. 2022), is marked bold. Orig. sum is coverage without normalization.

The results of our ablation analysis show the importance of our optimizations to make the computation feasible in practice. The naïve approach without any optimizations solves very few tasks, and only in two domains. Both optimizations are orthogonal, enabling them together maximizes coverage per domain for BW. The substantial differences with simple optimizations enabled suggests strong potential for future work aimed at reducing regression graph size.

When comparing FW and BW, we can see that both the positive and negative results from Figure 3 are reflected in coverage. Though, the margins are different. BW performs in Logistics, where it solves 90% of the instances, compared to the previous rate of just 17.5%. The increase in coverage for Childsnack and Blocksworld is modest in terms of (normalized) coverage, despite significant acceleration in heuristic computation speed. This is likely due to associated high branching factor and could likely be helped out with helpful actions. The proof of Proposition 4 hints a delete-relaxed plan extraction that could be used for extracting helpful action, to address the problem in future work.

When considering the results of our simple combination COMB, we can observe what we desired to achieve, it is able to select the faster method with insignificant overhead, taking advantage of this complementary behavior, and making it the top-performing h^{add} configuration in terms of coverage. This creates a new state-of-the-art in terms of pure-heuristic-performance on the lifted benchmark set. We suspect that this is likely to translate to non-pure heuristic approaches in the future once the helpful actions are available.

9 Related Work

There are other approaches that compute delete-relaxation heuristics at a lifted level. VHPOP (Younes and Simmons 2003) also computes h^{add} using lifted regression in the context of partial-order planning. However, a fundamental dif-

ference is that their approach performs grounding in every intermediate step (see Figure 2). We are not aware of any delete relaxation approach operating fully lifted like ours.

The heuristic by McDermott (1996) deviates additionally by greedily selecting a subset of variable replacements in each intermediate grounding step, instead of considering all possible matches. In Lemma 5, we demonstrate that the intermediate grounding does not alter the heuristic value. This insight helps to explain the connection between McDermott (1996) and h^{add}, which Bonet and Geffner (2001) left open: The difference in heuristic values is solely due to greedy selection.

While we are unaware of delete relaxation approaches operating fully lifted like ours, some approaches to solving planning tasks in general share a similar spirit. For example, Singh et al. (2023) regressively creates a disjunctive formulas, where each part of the disjunction roughly corresponds to a node in our tree, along with extra conditions for the deletes. However, they come with caveats. The main problem is ensuring replacements are valid, in a sense that there are no unintended deletes. This requires larger conditions, which are likely to be cyclic and so crucially harder for conjunctive query evaluation. Furthermore, computing actual plans requires considering that actions may achieve more than one atom in a single step. Our replacement of exactly one atom in the delete-relaxed case simplifies things considerably, as it reduces the potentially exponential number of replacement choices per action and node to a linear number. UCPOP (Penberthy and Weld 1992) can also be seen as an approach recursively creating a tree of first order logic formulas. The significant advantage is that the need for additional conditions is restricted to the case when there is a causal link. But, once again, contrary to our method, they apply intermediate grounding.

Conclusion

We analyzed strengths and limitations of lifted h^{add} computations in general and for the existing state-of-the-art computation. Motivated by our analysis, we introduced h^{LRadd}, a new lifted h^{add} computation. h^{LRadd} performs a lifted backward exploration, that avoids grounding entirely. Using it we introduce a new tractability island on tasks with low h^{add} value, regardless of the number of objects and predicate arity. In practice, h^{LRadd} is highly complementary to the traditional forward approach. Our experimental results demonstrate that a combination of both improves the state-of-the-art performance of pure-heuristic performance on hard-to-ground benchmarks. This further opens new research avenues for heuristics in lifted planning, e.g., computing other heuristics such as h^{FF} (Hoffmann and Nebel 2001; Corrêa et al. 2022) in similar ways.

Acknowledgments

We thank Augusto B. Corrêa and Daniel Fišer for their comments and feedback on earlier version of this paper.

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 232722074 – SFB 1102.

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases: The Logical Level*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN 0201537710.

Areces, C.; Bustos, F.; Dominguez, M.; and Hoffmann, J. 2014. Optimizing Planning Domains by Automatic Action Schema Splitting. In Chien, S.; Do, M.; Fern, A.; and Ruml, W., eds., *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*. AAAI Press.

Beeri, C.; Fagin, R.; Maier, D.; Mendelzon, A.; Ullman, J.; and Yannakakis, M. 1981. Properties of Acyclic Database Schemes. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC'81)*, 355–362.

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

Chandra, A. K.; and Merlin, P. M. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (STOC'77)*, 77–90.

Corrêa, A. B.; Francès, G.; Pommerening, F.; and Helmert, M. 2021. Delete-Relaxation Heuristics for Lifted Classical Planning. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS'21)*, 94–102. AAAI Press.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation Using Query Optimization Techniques. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS'20)*, 80–89. AAAI Press.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2022. The FF Heuristic for Lifted Classical Planning. In Honavar, V.; and Spaan, M., eds., *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI'22)*, 9716–9723. AAAI Press.

Corrêa, A. B.; and Seipp, J. 2022. Best-First Width Search for Lifted Classical Planning. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS'22)*. AAAI Press.

Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2): 75–88.

Fikes, R. E.; and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2: 189–208.

Flum, J.; Frick, M.; and Grohe, M. 2002. Query Evaluation via Tree-Decompositions. *Journal of the ACM*, 49(6): 716–752.

Graham, M. H. 1979. On the Universal Relation. *Technical Report, University of Toronto*.

Grohe, M.; Schwentick, T.; and Segoufin, L. 2001. When is the Evaluation of Conjunctive Queries Tractable? In *Proc. STOC*, STOC '01, 657–666. New York, NY, USA: Association for Computing Machinery. ISBN 1581133499.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173: 503–535.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 162–169. AAAI Press.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Höller, D.; and Behnke, G. 2022. Encoding Lifted Classical Planning in Propositional Logic. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS'22)*, 134–144.

Lauer, P.; Torralba, Á.; Fiser, D.; Höller, D.; Wichlacz, J.; and Hoffmann, J. 2021. Polynomial-Time in PDDL Input Size: Making the Delete Relaxation Feasible for Lifted Planning. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI'21)*, 4119–4126. IJCAI Organization.

Lauer, P.; Torralba, Á.; Höller, D.; and Hoffmann, J. 2025. Code and Appendix for paper: "Continuing the Quest for Polynomial Time Heuristics in PDDL Input Size: Tractable Cases for Lifted hAdd". doi:10.5281/zenodo.15323404.

Lipovetzky, N.; and Geffner, H. 2017. Best-First Width Search: Exploration and Exploitation in Classical Planning. In Singh, S.; and Markovitch, S., eds., *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*, 3590–3596. AAAI Press.

Masoumi, A.; Antoniazzi, M.; and Soutchanski, M. 2015. Modeling Organic Chemistry and Planning Organic Synthesis. volume 36 of *EPiC Series in Computing*, 176–195.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee.

McDermott, D. V. 1996. A Heuristic Estimator for Means-Ends Analysis in Planning. In Drabble, B., ed., *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS'96)*, 142–149. AAAI Press, Menlo Park.

Penberthy, J. S.; and Weld, D. S. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, 103–114. Morgan Kaufmann.

Ridder, B.; and Fox, M. 2014. Heuristic Evaluation Based on Lifted Relaxed Planning Graphs. In Chien, S.; Do, M.; Fern, A.; and Ruml, W., eds., *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*, 244–252. AAAI.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research*, 62: 535–577.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Singh, A.; Ramirez, M.; Lipovetzky, N.; and Stuckey, P. J. 2023. Lifted Sequential Planning with Lazy Constraint Generation Solvers. *arXiv preprint arXiv:2307.08242*.

Suda, M. 2016. Duality in STRIPS planning. In *ICAPS 2016 Workshop on Heuristics and Search for Domain-Independent Planning (HSDIP'16)*.

Vieille, L. 1986. Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In *Proceedings from the 1st International Conference on Expert Database Systems*, 253–267. Benjamin/Cummings.

Wichlacz, J.; Torralba, A.; and Hoffmann, J. 2019. Construction-Planning Models in Minecraft. In *ICAPS 2019 Workshop on Hierarchical Planning (HPlan'19)*, 1–5.

Yannakakis, M. 1981. Algorithms for Acyclic Database Schemes. In *Proceeedings of the Seventh International Conference on Very Large Data Bases (VLDB'81)*, 82–94. VLDB Endowment.

Younes, H. L. S.; and Simmons, R. G. 2003. VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research (JAIR)*, 20: 405–430.

Yu, C. T.; and Ozsoyoglu, M. 1979. An algorithm for tree-query membership of a distributed query. In *Proceedings of the Computer Software and The IEEE Computer Society's Third International Applications Conference (COMPSAC'79)*, 306–314. IEEE.