A Lifted Backward Computation of h^{add}

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

[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, lastname@cs.uni-saarland.de
[2]Aalborg University, Denmark, alto@cs.aau.dk

Abstract

Recent interest in solving planning tasks where full grounding is infeasible has brought attention to how to compute heuristics to guide the search at a lifted level. h^{add} is a well-understood heuristic for classical planning. Methods to compute h^{add} perform a forward fix-point computation, as this takes polynomial time with respect to the grounded representation. However, this computational efficiency does not carry over to lifted planning tasks, where it is EXPTIME-complete to compute h^{add}.

In this paper, we introduce a novel approach for computing h^{add} on lifted planning tasks. Our approach proceeds backwards, constructing a fully lifted regression graph whose nodes are assessed using conjunctive query evaluation. We provide a complexity analysis and show that our backward computation is complementary to the traditional forward computation. Our empirical evaluation confirms that there are significant differences in the performance of both methods in each domain. Overall, both methods are very complementary, and their combination advances the state-of-the-art in lifted h^{add} computations.

1 Introduction

It has been known since a long time that the grounding preprocesses commonly used in state-of-the-art planning systems can be a performance bottleneck when the size of the grounded representation is large (Erol, Nau, and Subrahmanian 1995; Helmert 2009; Areces et al. 2014; Masoumi, Antoniazzi, and Soutchanski 2015). In the last years, this challenge has been taken up again with a drive to transfer methods from heuristic search planning to the lifted-planning setting, where grounding is avoided to the extent possible. In particular, Corrêa et al. (2020) introduced an effective forward search by linking state expansion to answering conjunctive queries over databases. Several works have devised heuristic functions for the lifted setting, including variants of the relaxed plan heuristic (Lauer et al. 2021; Corrêa et al. 2022), landmark heuristics (Wichlacz, Höller, and Hoffmann 2022; Wichlacz et al. 2023), and h^{add} (Corrêa et al. 2021).

We focus on h^{add} (Bonet and Geffner 2001), which is part of the canon of competitive delete relaxation heuristics. The original computation is a grounded fix point forward computation that assigns a value to each propositional-fact. It starts at the current state, and iteratively updates fact values if a cheaper achiever for a fact is discovered. This takes time polynomial in the size of the grounded representation. Corrêa et al. (2021) extended this idea to the lifted level. They adapt the state-of-the-art grounding technique (Helmert 2009) to generate *one* cheapest achiever on demand from the lifted representation without generating all actions that can achieve it. The computation is an essential part of the top performing configurations on the standard lifted planning benchmarks. This includes the integration into h^{FF} (Hoffmann and Nebel 2001; Corrêa et al. 2022) as well as combining it with best-first width search (Lipovetzky and Geffner 2017; Corrêa and Seipp 2022).

To better understand the strengths and limitations of previous h^{add} computation methods, we analyze the complexity of computing h^{add} at a lifted level. We prove that the general case is EXPTIME-complete, making an exponential blow up unavoidable. The blow up in Corrêa et al. (2021) arises from tagging each fact with an h^{add} value, which can be exponentially many. We observe that h^{add} often remains polynomial-time computable when grounding all facts is not. But, so far there is no alternative lifted h^{add} computation that works without grounding facts.

Here we fill this gap. We propose an alternative lifted h^{add} computation. Our approach works backwards (using *regression*), gradually building new conditions represented by lifted atoms until one is satisfied. Conditions are built starting from the goal, by replacing unsatisfied parts with lifted action preconditions that can satisfy them. This allows to only generate *one* grounded certificate for some lifted condition and so eliminates the tie to grounding all facts. Variants of lifted regression appeared for various similar purposes in the literature (McDermott 1996; Penberthy and Weld 1992; Vieille 1986; Younes and Simmons 2003). However, these variants ground more than our approach, either through repeated (partial) variable substitution or a final enumeration to combine results. Thus, they encounter a similar bottleneck as grounding all facts. We identify lifted planning tasks where our computation is guaranteed to be polynomial. Further, we introduce optimizations to make the computation practical.

We run an empirical evaluation on the standard lifted planning benchmark set and compare our new h^{add} computation to the state-of-the-art lifted forward computation. Like our theoretical analysis indicates, the empirical results

show that both methods are highly complementary on a per-domain basis. Our new method outperforms the forward computation in some domains, while the latter remains dominant in overall performance. We combine both approaches, leveraging their individual strengths, thereby advancing the state-of-the-art in lifted h^{add} computation.

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 denoted as $X(L)$. $p(\vec{x})$ is called grounded 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 replacing its variables with objects. We formalize replacement similar to McDermott (1996) by defining variable replacement functions $\mathrm{match}(\delta) := \{\theta : X(\delta) \to \mathcal{O}\}$ to replace the variables in some structure $\delta \in \mathcal{P}^X \cup \mathcal{A} \cup 2^{\mathcal{P}^X}$ with objects. $\theta \in \mathrm{match}(\delta)$ is applied to δ by replacing all occurrences of a variable according to θ. This is denoted by $\theta(\delta)$. E.g. $\theta(p(\vec{x}))$ for a lifted atom or $\theta(a)$ for an action. $X(\theta)$ denotes the domain of θ. This definition extends to variable remapping $\sigma : X(\delta) \to \mathcal{O} \cup X$, that are not guaranteed to ground but can also replace variables by other variables. A grounded task is obtained by replacing the actions a by all possible 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 $\mathrm{progr}(\theta_n(a_n), \mathrm{progr}(..., \mathrm{progr}(\theta_1(a_1), s)))$ by $\mathrm{progr}(\theta_1(a_1), ..., \theta_n(a_n), s)$. A plan is an action sequence $\theta_1(a_1), \ldots, \theta_n(a_n)$ so that $\mathrm{progr}(\theta_1(a_1), \ldots, \theta_n(a_n), \mathcal{I}) \supseteq \Gamma$. A task is solved by finding a plan.

2.1 Grounded Computation of h^{add}

Bonet and Geffner (2001) introduced h^{add} together with the delete relaxation Π^+ of a planning task Π, which is obtained by removing the delete lists of all actions. The main idea is to allow achieving grounded atoms independently to simplify the planning problem. We define the achievers for a grounded atom as: $\mathrm{ach}(\{p(\vec{o})\}) = \{\theta(a) \in \mathcal{A}^{\mathcal{O}} \mid p(\vec{o}) \in add(\theta(a))\}$. And define h^{add} as the point-wise greatest function fulfilling:

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

In practice h^{add} is commonly computed by a bottom up dynamic programming approach that stores the cheapest value $h^{add}(s, \{p(\vec{o})\})$ for all $p(\vec{o}) \in \mathcal{P}^{\mathcal{O}}$.

2.2 Satisfiability and Query Evaluation

If we always generated the entire set $\mathrm{match}(\delta) := \{\theta : X(\delta) \to \mathcal{O}\}$ for structures $\delta \in \mathcal{P}^X \cup \mathcal{A} \cup 2^{\mathcal{P}^X}$, one could also just ground the task. 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 if there exists a tree with nodes L so that for any $x \in X(L)$ and $p_1(\vec{x}_1), p_2(\vec{x}_2) \in L$ it holds that $x \in X(p_1(\vec{x}_1))$ and $x \in X(p_2(\vec{x}_2))$ iff there is a path in the tree 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 $X(Y)$ restricted in size, then the output is guaranteed to be bounded w.r.t.

the lifted task representation and Yannakakis algorithm can evaluate $\pi_Y(\text{match}(s, L))$ in polynomial time. Many algorithms, associate a weight $w : s \to \mathbb{R}$ with every element of a state $s \in S$. This could e.g. be the h^{add} cost of each atom in the state. In this case, Yannakakis algorithm, or other join algorithms, can be adapted to compute the cheapest cost $\min_{\theta \in \text{match}(s,L)} \sum_{p(\vec{x}) \in L} w(\theta(p(\vec{x})))$.

$\mathcal{P}^{X+}$ 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 amount of elements in multisets is irrelevant in this context.

3 The Complexity of Computing Lifted h^{add}

In this section we identify the complexity of computing h^{add} on lifted planning tasks and link this to advantages and limitations of the lifted forward computation. The result emphasizes the necessity of considering different lifted h^{add} computations. Due to space constraints, we present proof sketches throughout the whole paper. Full proofs are provided in the Appendix.[1]

Theorem 1. *Computing h^{add} on a lifted planning task 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}. □

The membership proof computes h^{add} at the grounded level, indicating that lifted h^{add} computation is generally as hard as grounding. This very closely aligns with Corrêa et al.'s limitation, as it is an adaptation of the grounding algorithm by Helmert (2009) but has no general guarantee to work better. In a nutshell its main performance advantage closely aligns with the efficiency of join algorithms that produce a limited output. Instead of considering all grounded actions, it can potentially determine only a subset of actions to tag all delete relaxed reachable atoms with their h^{add} cost. However, this also means that the algorithm may consider all grounded atoms, which we will identify as its main limitation. We will refrain from revisiting the complete approach. Instead, we want to view this limitation in a more general setting, for any approach that assigns the h^{add} value to each atom separately. To the best of our knowledge, any known lifted progression, even outside the scope of planning, contains an intermediate grounding step to obtain grounded atoms. Thus we will associate this limitation with lifted forward computations for h^{add} in general.

Before understanding when this becomes a limitation we will emphasize the potential strength of such approaches. This will later help to highlight that forward computations can complement our backward computation. We do so by constraining the two task properties influencing their runtime exponentially. This is: (1) The amount of delete-relaxed reachable atoms, by constant predicate arity. And (2) the hardness of conjunctive query evaluation, by considering acyclic tasks. These constraints align with insights from prior research. In Lauer et al. (2021), tractability is achieved by constraining predicate arity to 1, Corrêa et al. [2020; 2023] exploit acyclic planning tasks.

Theorem 2. *Computing h^{add} on an acyclic lifted planning task with predicate arity at most $k \in \mathbb{N}$ is in PTIME.*

Proof sketch. The restriction to limited predicate arity allows to ground all atoms in polynomial time and space. Thus we adapt the dynamic programming approach from the grounded setting by considering grounded atoms and lifted actions. We generate the currently cheapest achievable atoms using Yannakakis algorithm. The restriction to acyclic preconditions allows to do this in polynomial time. Doing this iteratively, in a uniform cost manner, assigns the cost a polynomial amount of times. □

Applying only one constraint at a time does not guarantee a polynomial runtime. This is evidenced by the NP-hardness of delete-relaxed plan existence under limited predicate arity (Lauer et al. 2021) for (1). And for (2), as any planning task can become acyclic by increasing predicate arity, as we prove in the following Proposition.

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. □

This marks an interplay between predicate arity and acyclicity. It further highlights why in certain tasks it can be detrimental to generate all delete relaxed reachable atoms. Here a forward exploration would create all grounded atoms for p_a which grow exponentially w.r.t. predicate arity.

To illustrate the existence of interesting yet simplistic domains with this limitation, we consider a simplified version of the Childsnack domain from the lifted planning benchmark set, shown in Figure 2. 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 $\text{sandwich}(o_{c_1}, \ldots, o_{c_n})$. This grows exponentially in n, making the forward evaluation infeasible. Indeed, as the arity of the sandwich predicate scales, Childsnack falls outside the tractability case of Theorem 2.

[1] fai.cs.uni-saarland.de/lauer/manuscripts/hsdip2024-tr.pdf

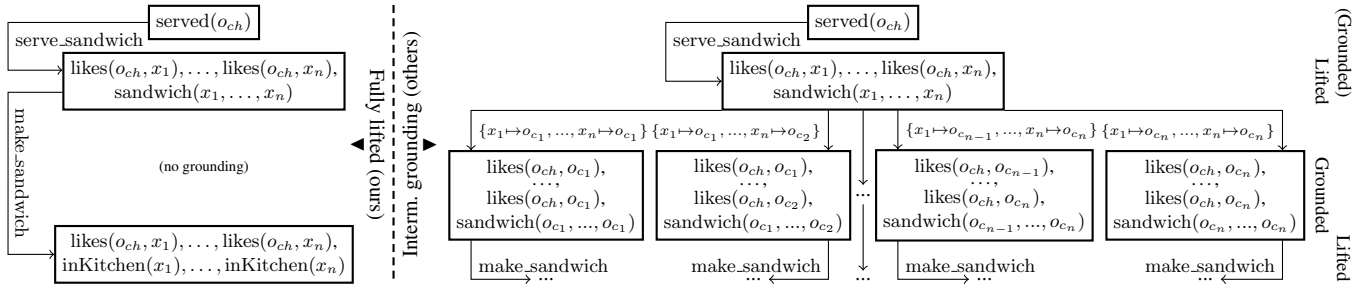

Figure 1: The lext regression graph (left). Related approaches with exponential node explosion in one intermediate grounding step (right).

4 A Lifted Regressive Formulation for h^{add}

In this section we will formulate h^{add} in an alternative way. 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 any 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 that can achieve it. This ensures that if $s \models L$ for any node, a (relaxed) plan exists. When each node is linked to the total cost of actions along its creation 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 1 shows the lext tree for our running example, with one goal $\mathrm{served}(o_{ch})$, to the left. The only action that may achieve the goal $\mathrm{served}(o_{ch})$ is serve_sandwich. We replace the atom $\mathrm{served}(o_{ch})$ with the precondition of serve_sandwich, replacing the first parameter to match $\mathrm{served}(o_{ch})$. 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 $\mathrm{sandwich}(x_1, \ldots, x_n)$ is subsequently replaced by make_sandwich in the next step.

make_sandwich($x_{c1} \ldots x_{cn}$ – content):
pre : {inKitchen(x_{c1}), ..., inKitchen(x_{cn})}
add : {sandwich($x_{c1}, \ldots, x_{cn}$)}
del : {inKitchen(x_{c1}), ..., inKitchen(x_{cn})}
serve_sandwich(x_{ch} – child $x_{c1} \ldots x_{cn}$ – content):
pre : {likes(x_{ch}, x_{c1}), ..., likes(x_{ch}, x_{cn}), sandwich($x_{c1}, \ldots, x_{cn}$)}
add : {served(x_{ch})}
del : {sandwich($x_{c1}, \ldots, x_{cn}$)}

Figure 2: Actions for the Childsnack running example.

The tree to the right of Figure 1 illustrates what would happen 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 representations 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 replacing grounded actions on the path to a satisfied node, compare Suda (2016).

We will now formalize the approach. We start with the part of lext which remaps the parameters of the action s.t. the add element matches exactly the description of some $p(\vec{x}) \in L$ for a given $L \in \mathcal{P}^{X+}$. 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:*

(1) $p(\vec{x}) \in add(\sigma(a))$
(2) $X(\sigma(a)) \cap X(L) = X(p(\vec{x}))$
(3) $\sigma_{|X(a) \setminus X(add(a))}$ is injective

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

Condition (1) ensures that the parameters of the add element are remapped to match the lifted atom it shall replace. Condition (2) and (3) combined ensure that variables that are not part of the add element, are each replaced by a unique new variable that is not already part of $L \in \mathcal{P}^{X+}$ to not additionally restrict variable substitutions.

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

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

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

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

Note that the choice $\sigma \in \text{VarRemap}(L, p(\vec{x}), a)$ is not uniquely determined. 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 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:*

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

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

5 Equality to h^{add}

Before describing in detail how to compute h^{LRadd}, we will prove that h^{add} and h^{LRadd} produce the same heuristic values. We consider two separate cases for $s \in S$: $h^{add}(s) = \infty$ and $h^{add}(s) \neq \infty$.

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 $s \models L_0$ (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 $h^{add}(s) = \infty$. □

Note that the proof hints a delete-relaxed plan extraction, though for this paper, our focus remains on a pure computational analysis of h^{add}. It seems reasonable to assume that the advantages we observe for h^{add} would transfer to a delete relaxed plan extraction.

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 1.

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 $s \models L_0$. Let θ^* be a variable replacement under which $s \models L_0$. Then, $h^{LRadd}(s, L) = h^{LRadd}(s, \theta^*(L)) \geq \min_{\theta \in \text{match}(L)} h^{LRadd}(s, \theta(L))$. □

The proof highlights the advantage of h^{LRadd} over the naïve grounded regression as it entails an exponential decrease of nodes in the running example. This result further provides an important insight to understand the connection between h^{add} and other lifted heuristics (McDermott 1996), as we discuss in Section 9. 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. Recall that the value of h^{LRadd} corresponds to the cost of one cheapest lext-path. W.l.o.g. assume $G = \{p_1(\vec{o}_1), p_2(\vec{o}_2)\}$. To show $\leq$ we split any lext-path from G into two parts: One path for the (recursive) replacements of $p_1(\vec{o}_1)$ and the rest corresponding to the replacements of $p_2(\vec{o}_2)$. The splits add up to the same cost. A split in this way is possible as $p_1(\vec{o}_1), p_2(\vec{o}_2)$ are grounded and so do not impose variable constraints onto each other. To show $\geq$ we argue that any two lext-paths for $\{p_1(\vec{o}_1)\}$ and $\{p_2(\vec{o}_2)\}$, can be combined by rewriting the variables of the $p_2(\vec{o}_2)$ path to exclude any variables from the $p_1(\vec{o}_1)$ path, and then append them. □

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. 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 previously used benchmarks. The runtime of h^{LRadd} is bounded by the amount of explored nodes times the runtime for the most expensive satisfiability check. To determine a bound on the nodes explored 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 $E_{s,i+1} := \{L' \mid (a, L') \in \mathrm{lext}(L), L \in E_{s,i}\}$. And set $E_s := \bigcup_{i=0}^{h^{add}(s)} E_{s,i}$.*

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 amount 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 encounter 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. *Computing h^{add} on a lifted planning task 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 is in PTIME.*

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. □

The restriction in Theorem 9 fits very well into the picture of planning tasks that are easy to solve due to having short plans, but have a lot of delete-relaxed reachable ground atoms. This is one of the motivations described by Ridder and Fox (2014) or Lauer et al. (2021). An example where forward exploration leads to exponential blow-up due to increasing predicate arity and number of objects is the lifted Childsnack variant, corresponding to our running example.

Of course, our evaluation is expected to outperform the forward evaluation when the latter encounters exponential blow-up, while our backward approach maintains a polynomial runtime. A more interesting observation is that this expectation can extend to situations where both Theorems 2 and 9 apply, meaning both approaches are guaranteed to be polynomial. The reason is that even with restricted predicate arity, the forward evaluation often struggles with a hugely increasing amount of objects, due to the large number of grounded atoms generated. When the h^{add} values for subgoals are fixed to a small constant in a domain, our approach excels by avoiding the generation of additional grounded atoms. The advantage is that we only need a small, constantly-bounded, number of conjunctive query evaluations. Under acyclicity, this evaluation reduces tables, that represent at most the current state, preventing to generate new atoms. In other words, our approach scales independently of the number of objects, unlike the forward evaluation.

The lifted benchmark set introduced by Lauer et al. (2021) contains examples illustrating precisely this behavior. In Blocksworld all blocks are initially on the table. Thus in the delete-relaxed setting one stack operation per subgoal suffices, bounding h^{add} by a small constant. The backward exploration essentially checks whether the few required blocks to achieve the goal can be stacked. The forward exploration, however, would generate a quadratic amount of stacks that correspond to all possible combinations. Another example is Logistics where each package can be delivered in a constant amount of steps, regardless the number of packages, trucks, or cities. Then again our h^{add} value is bounded by a small constant. Scaling up the map size or the number of vehicles leads to the forward evaluation exploring the navigation of all vehicles through all locations to make any deliveries.

There are further examples within the commonly used IPC benchmark set. Gripper is a good example, where the robot picks up a ball, moves it, and places it in another room. Similar to Blocksworld, h^{LRadd} can easily check that a concrete ball can be moved, while the forward evaluation moves all objects, regardless whether they need to be moved.

7 Making the Computation Practicable

While Theorem 9 highlights a big strength of h^{LRadd}, the straightforward realization outlined in section 6 can be impractical. 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. The main idea is that 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. This allows to enforce the replacement immediately. □

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, as based on Yannakakis and GYO, produce L_S as a byproduct when L is unsatisfiable. As atoms are combined incrementally, we take L_S as the atoms 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 sets of 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 $h^{LRadd}(s, L_1) + h^{LRadd}(s, L_2) = h^{LRadd}(s, L)$.*

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

Table 1: Normalized coverage (percentage of solved instances) of different h^{add} computations: our approach with each optimization enabled/disabled (BW), the forward lifted computation (FW) by Corrêa et al. (2021), and their combination (COMB).

Proof sketch. This is a special case of Thm 10. First enforce all replacements of L_1 until satisfied and then of L_2. □

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

8 Experiments

We implemented h^{LRadd} with the optimizations described in section 7 within the Powerlifted Planner (Corrêa et al. 2020). We coupled our heuristic and the lifted h^{add} variant of Corrêa et al. (2021) for comparision 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. The source code of our implementation will be publicly available.

Table 1 presents a comparison of normalized coverage, i.e. percentages of tasks solved per domain, as used in Höller and Behnke (2022).[2] We perform 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). We verified correctness the correctness of our implementations by confirming that the expansions and initial heuristic value FW. As both approaches turned out to be very complementary, we also introduce a simple combination (COMB) that automatically picks which method to use in each instance. To do

[2]Normalizing the coverage reduces the influence of the huge disparity of instances per domain in this benchmark set.

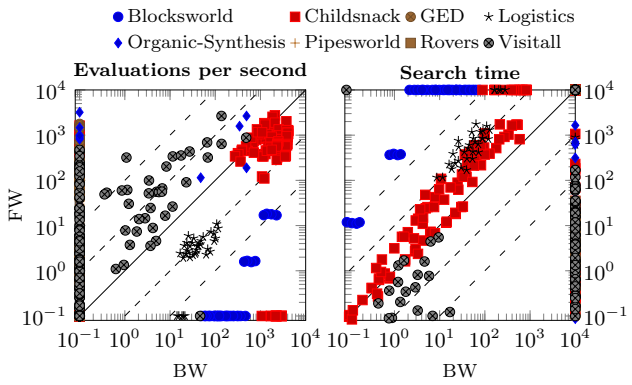

Figure 3: Comparison of FW and BW in terms of evaluations per second and search time. Search time is measured up to the last common f-layer. Data points in the extremes represent cases where one of the methods failed to compute the heuristic value for the initial state.

that, 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.

The results of our ablation analysis show the importance of our optimizations in order 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, so enabling them together maximizes coverage in each domain for BW. The substantial differences between enabled and disabled simple optimizations further underlines the potential for this approach to excel with further investigation aimed at reducing regression graph size.

When comparing FW and BW, we see that they 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 is particularly impressive in Logistics, where BW solves 90% of the instances, compared to the previous rate of just 17.5%. On the other hand, we also observe a notable decrease in coverage for the remaining five domains, with three of them remaining entirely unsolved. This aligns with our earlier analysis of complexity, as these domains exhibit longer delete-relaxed plans, and comparably fewer delete-relaxed reachable facts.

This indeed presents complementary picture we expected to find. However, this does not render h^{LRadd} useless, as it is easy to detect when it pays off or not. Our simple combination COMB 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.

While the increase in coverage for Childsnack and Blocksworld may appear modest in terms of (normalized) coverage, there is a significant acceleration in heuristic computation speed. This is clearly demonstrated in Figure 3, which compares the number of evaluations per second and search time of FW and BW. In order to compare the performance also on tasks that are not entirely solved, we compare the performance until the last common f-layer. Note that both approaches compute the same heuristic values, so the comparison is always up to expanding the exact same set of states.

In this comparison, we can see that the speed up in heuristic computation is of one order of magnitude in Logistics and more than two orders of magnitude in Blocksworld. Additionally, the data points for Blocksworld reinforce our theoretical analysis. They reveal three distinct plateaus of data points, each corresponding to a significantly increased number of delete-relaxed reachable atoms needed to compute the h^{add} value with the forward method.

On the other hand, we also observe that there are many instances where the methods (specially BW), fail to even compute the heuristic of the initial state. This highlights the importance of considering multiple methods to compute these heuristics at a lifted level.

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 difference is that their approach performs grounding in every intermediate step (see Figure 1). 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. And of course, there are less computational guarantees, as in grounded planning tasks the forward exploration always runs in polynomial time.

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. Whereas our replacement of exactly one atom in the delete relaxed case simplifies things considerably. 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 introduced h^{LRadd}, a new method to compute the h^{add} heuristic at a lifted level. h^{LRadd} performs a lifted backward exploration, where each node corresponds to a conjunctive query. This avoids grounding entirely, introducing a new tractability island on tasks with low h^{add} value, regardless of the amount of objects and predicate arity. In practice, h^{LRadd} is highly complementary to the traditional forward approach, and experimental results demonstrate that a combination of both can achieve state-of-the-art performance in lifted h^{add} computation across various domains. 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.

Acknowledgment

We thank Augusto B. Corrêa for very insightful discussions on the paper and his related work.

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, 1st edition. ISBN 0201537710.

Areces, C.; Bustos, F.; Dominguez, M.; and Hoffmann, J. 2014. Optimizing Planning Domains by Automatic Action Schema Splitting. In *Proc. ICAPS*.

Beeri, C.; Fagin, R.; Maier, D.; Mendelzon, A.; Ullman, J.; and Yannakakis, M. 1981. Properties of Acyclic Database Schemes. In *Proceedings of STOC '81*.

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

Chandra, A. K.; and Merlin, P. M. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *9th ACM Symposium on the Theory of Computation*.

Corrêa, A. B.; Francès, G.; Pommerening, F.; and Helmert, M. 2021. Delete-Relaxation Heuristics for Lifted Classical Planning. In *Proceedings of the 31st ICAPS*.

Corrêa, A. B.; Hecher, M.; Helmert, M.; Longo, D. M.; Pommerening, F.; and Woltran, S. 2023. Grounding Planning Tasks Using Tree Decompositions and Iterated Solving. In *ICAPS*.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation Using Query Optimization Techniques. In *ICAPS*.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2022. The FF Heuristic for Lifted Classical Planning. In *Proceedings of the 36th AAAI*.

Corrêa, A. B.; and Seipp, J. 2022. Best-First Width Search for Lifted Classical Planning. In *ICAPS*.

Eiter, T.; Faber, W.; Fink, M.; and Woltran, S. 2007. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence*, 51(2).

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

Flum, J.; Frick, M.; and Grohe, M. 2002. Query Evaluation via Tree-Decompositions. *J. ACM*, 49(6).

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 *Proceedings of STOC '01*. New York, NY, USA.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *AI*, 173.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR*, 14.

Höller, D.; and Behnke, G. 2022. Encoding Lifted Classical Planning in Propositional Logic. *Proceedings of ICAPS*.

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 IJCAI*.

Lipovetzky, N.; and Geffner, H. 2017. Best-First Width Search: Exploration and Exploitation in Classical Planning. In *AAAI*.

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

McDermott, D. V. 1996. A Heuristic Estimator for Means-Ends Analysis in Planning. In *Proceedings of the 3rd AIPS*.

Penberthy, J. S.; and Weld, D. S. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *KR*.

Ridder, B.; and Fox, M. 2014. Heuristic Evaluation Based on Lifted Relaxed Planning Graphs. In *ICAPS*.

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 *Proceedings of 8th Workshop on Heuristics and Search for Domain-independent Planning (HSDIP) at ICAPS'16*.

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

Wichlacz, J.; Höller, D.; Fiser, D.; and Hoffmann, J. 2023. A Landmark-Cut Heuristic for Lifted Optimal Planning. In *ECAI*.

Wichlacz, J.; Höller, D.; and Hoffmann, J. 2022. Landmark Heuristics for Lifted Classical Planning. In *IJCAI*.

Yannakakis, M. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*.

Younes, H. L. S.; and Simmons, R. G. 2003. VHPOP: Versatile Heuristic Partial Order Planner. *JAIR*, 20.

Yu, C. T.; and Ozsoyoglu, M. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC*.