The FF Heuristic for Lifted Classical Planning

Augusto B. Corrêa[1], Florian Pommerening[1], Malte Helmert[1], Guillem Francès[2]

[1]University of Basel, Switzerland
[2]Universitat Pompeu Fabra, Spain
{augusto.blaascorrea,florian.pommerening,malte.helmert}@unibas.ch
guillem.frances@upf.edu

Abstract

Heuristics for lifted planning are not yet as informed as the best heuristics for ground planning. Recent work introduced the idea of using Datalog programs to compute the additive heuristic over lifted tasks. Based on this work, we show how to compute the more informed FF heuristic in a lifted manner. We extend the Datalog program with executable annotations that can also be used to define other delete-relaxation heuristics. In our experiments, we show that a planner using the lifted FF implementation produces state-of-the-art results for lifted planners. It also reduces the gap to state-of-the-art ground planners in domains where grounding is feasible.

Introduction

Planning is one of the oldest subfields of Artificial Intelligence. A planning task defines the current state of the world and a set of *actions*. Applying an action changes the state of the world. The objective of planning is to find a sequence of actions leading to a state that satisfies a goal condition.

Planning tasks are usually described in a *lifted representation* (McDermott 2000). While it was common to work directly on lifted representations (e.g., McDermott 1996, Reiter 2001), research in the last two decades mostly focused on *ground planning* where the task is grounded to a propositional description that is then used to solve this task. *Heuristic search* (Bonet and Geffner 2001), in particular, has been very successful for ground planning. The main idea here is to search for a solution in the space of all applicable action sequences. Using a heuristic, the planner focuses the search on promising sequences and thus speeds up the process.

Grounding a task sometimes has high costs, even if the final propositional representation is not that large. For this reason *lifted planners* – those that work directly on the lifted representation – are seeing renewed interest (Corrêa et al. 2020; Lauer et al. 2021; Horčík and Fišer 2021). To benefit from the advances in ground planning, in particular in the area of heuristic search, we need strong heuristics that can be computed for lifted representations.

The *FF heuristic* h^{FF} (Hoffmann and Nebel 2001) is part of state-of-the-art ground planners even 20 years after its invention. It is based on the *delete-relaxation* of a planning task, where any fact that is made true by an action remains true forever. This estimate is usually well informed and thus gives good guidance for heuristic search planners.

We show how to compute h^{FF} on lifted representations, making it unnecessary to ground the task. Our work is based on using Datalog programs to compute delete-relaxation heuristics (Corrêa et al. 2021). We introduce the notion of an *annotated Datalog program*, where atoms and rules are associated with instructions. After evaluating a query, we execute the instructions of all atoms and rules used to derive this query. This allows us to collect a relaxed plan and compute h^{FF}. Furthermore, we show how to transform annotated Datalog programs in order to make them cheaper to evaluate. Our transformations can simulate the optimizations introduced by Corrêa et al. (2021) and have the advantage of being independent of previous transformations. In our experiments, we show that a planner using the lifted h^{FF} heuristic achieves state-of-the-art performance among lifted planners and is competitive with Fast Downward (Helmert 2006) using a ground implementation of the heuristic.

Background

In this paper, planning languages and Datalog programs assume a *function-free logical vocabulary* over an infinite set of *variables* $\mathcal{V}$, a finite set of *constants* $\mathcal{C}$, and a finite number of *predicate symbols* $\mathcal{P}$. A *term* t is either a variable or a constant. Each predicate symbol has a non-negative arity (in particular, this may be zero). Given an n-ary predicate $P \in \mathcal{P}$ and a tuple of terms $\mathbf{t} = \langle t_1, \ldots, t_n \rangle$, we call $P(\mathbf{t})$ an *atom*. If $\mathbf{t}$ contains a variable, then $P(\mathbf{t})$ is a *lifted atom*, otherwise it is a *ground atom*. We define $Atoms$ as the set of all possible atoms. *Grounding* an atom $P(\mathbf{t})$ replaces all variables in $\mathbf{t}$ by constants according to a *substitution* $\sigma : \mathcal{V} \to \mathcal{C}$.

Classical Planning

With predicate symbols $\mathcal{P}$ and constants $\mathcal{C}$, a *lifted planning task* is a tuple $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, s_0, \mathit{goal} \rangle$. The finite set $\mathcal{A}$ represents the *action schemas*. An action schema $A \in \mathcal{A}$ consists of a *cost* $c(A) \in \mathbb{R}_0^+$, and of three sets of atoms: the *precondition* $\mathit{pre}(A)$, the *add list* $\mathit{add}(A)$, and the *delete list* $\mathit{del}(A)$. The *parameters* of A are the set of variables occurring in any atom of $\mathit{pre}(A) \cup \mathit{add}(A) \cup \mathit{del}(A)$ in some fixed order. An action schema A with no parameters is a *ground action* (sometimes called just *action*). If action schema A

has parameters $\langle X_1, \ldots, X_n \rangle$ and it is ground with constants $\langle C_1, \ldots, C_n \rangle$ then we write $A(C_1, \ldots, C_n)$ for the ground action. A planning task Π where all action schemas are ground is a *ground planning task*. A *state* $s \subseteq Atoms$ is a set of ground atoms with the interpretation that exactly these atoms are true. The *initial state* s_0 is a state containing the atoms that are initially true. The ground atom $goal \in Atoms$ should be reached to solve the task. (Larger conjunctive goals φ can compiled away by adding a zero-cost action with precondition φ and add list $\{goal\}$.)

The precondition *pre*(A) of a ground action A is *satisfied* in a state s iff *pre*$(A) \subseteq s$. In this case, we say that action A is *applicable*. The *successor state* s' of an applicable action A is defined as $s' = (s \setminus$ *del*$(A)) \cup$ *add*(A). A sequence of actions $A_1, \ldots, A_n$ is applicable in a state s_0 if there are states $s_1, \ldots, s_n$ where A_i is applicable in s_{i-1} and its application leads to s_i. A *goal state* s_* is a state where $goal \in s_*$. An applicable sequence of actions $\pi = \langle A_1, \ldots, A_n \rangle$ from the initial state s_0 to some goal state is called a plan and has cost $c(\pi) = \sum_{i=1}^{n} c(A_i)$. The task is *solvable* if a plan exists.

A function that estimates the distance between states and their closest goal state is a *heuristic*. One prominent family of heuristics are the *delete-relaxation heuristics* (Bonet and Geffner 2001). The delete-relaxation of a task $\Pi = \langle \mathcal{P}, O, \mathcal{A}, s_0, goal \rangle$ is the task Π^+ obtained by redefining *del*$(A) = \emptyset$ for all action schemas $A \in \mathcal{A}$. By ignoring the delete lists, applying an action in a state s only adds more ground atoms to the state. Thus, all actions that are applicable in s remain applicable in any successor state of s. The relaxation Π^+ is an overapproximation of Π: any atom *reachable* (i.e., achieved by any sequence of actions) in Π is also reachable for Π^+. The converse is not always true.

One way of computing delete-relaxation heuristics is to use the cost of a *relaxed plan* (a plan for Π^+) as heuristic value (e.g., Hoffmann and Nebel 2001, Keyder and Geffner 2008). We denote the cost of an *optimal relaxed plan* as h^+. Delete-relaxation heuristics that approximate h^+ tend to compare favorably with other heuristics (Hoffmann 2005; Betz and Helmert 2009).

Datalog Programs

A *Datalog rule* r has the form $P \leftarrow Q_1, \ldots, Q_m$, for $m \geq 1$. It consists of an atom $head(r) = P$ called the *head* of the rule and a set of atoms $body(r) = \{Q_1, \ldots, Q_m\}$ called its *body*. We write *vars*(r) for the set of variables in the body and head. A *Datalog program* $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ is a pair where the set of *facts* $\mathcal{F}$ is a set of ground atoms, and set of *rules* $\mathcal{R}$ is a set of Datalog rules.

Given a Datalog rule $r \in \mathcal{R}$, $r = P \leftarrow Q_1, \ldots, Q_m$ with *vars*$(r) = \{v_1, \ldots, v_n\}$, we define $r_\forall = \forall v_1, \ldots, v_n. Q_1 \wedge \ldots \wedge Q_m \rightarrow P$. The *canonical model* of a Datalog program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ is the maximal set $\mathcal{M}$ of ground atoms such that $\mathcal{F} \cup \{r_\forall \mid r \in \mathcal{R}\} \models \mathcal{M}$. Each Datalog program has a unique canonical model (Abiteboul, Hull, and Vianu 1995). In the case where the set of rules $\mathcal{R}$ is part of the input, computing $\mathcal{M}$ is EXPTIME-complete (Dantsin et al. 2001).

For a rule $r \in \mathcal{R}$ we use $Ground(r)$ for the set of rules obtained by applying all possible substitutions of variables in r with constants in $\mathcal{C}$. We also define $Ground(\mathcal{R}) = \bigcup_{r \in \mathcal{R}} Ground(r)$.

A derivation is a sequence of facts from $\mathcal{F}$ and ground rules from $Ground(\mathcal{R})$ where all atoms in the body of a rule either occur earlier in the derivation or are heads of rules that appear earlier in the derivation. A fact P in a derivation demonstrates $P \in \mathcal{M}$ while a rule r demonstrates that $head(r) \in \mathcal{M}$. We assume that each atom is derived at most once in a derivation. If the last atom derived is P, we call the derivation a *derivation of* P. An *achiever choice function* $f : \mathcal{M} \setminus \mathcal{F} \rightarrow Ground(\mathcal{R})$ maps atoms P to ground rules r with $head(r) = P$ that occur in a derivation. Given an achiever choice function, we can construct a derivation of an atom by back-chaining through the atoms in achiever bodies. For each atom in $\mathcal{M} \setminus \mathcal{F}$ that was not seen earlier, we recursively add its derivation to the start of the sequence.

Computing the Canonical Model There are several algorithms to compute the canonical model $\mathcal{M}$ of a Datalog program (see Abiteboul, Hull, and Vianu 1995). In this paper, we will use the algorithm introduced by Helmert (2009). The idea is to build the canonical model incrementally.

The algorithm starts with a queue $\mathcal{Q} \coloneqq \mathcal{F}$ and $\mathcal{M} \coloneqq \emptyset$. It then removes atoms from $\mathcal{Q}$, one by one, until $\mathcal{Q}$ is empty. Whenever an atom Q is removed, the algorithm checks whether it is already in the canonical model. If so, the algorithm continues with the next atom in the queue. Otherwise, Q is added to $\mathcal{M}$ and the algorithm generates the set of all ground rules $r \in Ground(\mathcal{R})$ where $\mathcal{M} \models body(r)$ and $Q \in body(r)$. For each such rule, $head(r)$ is added to $\mathcal{Q}$. The algorithm implicitly constructs a derivation with an achiever choice function that maps $head(r)$ to r. Note that the achiever choice function depends on how $\mathcal{Q}$ is ordered.

Datalog-Based Heuristics

For a given planning task $\Pi = \langle \mathcal{P}, O, \mathcal{A}, s_0, goal \rangle$ and a given state s, Corrêa et al. (2021) encode Π^+ as a Datalog program $\mathcal{D}_s = \langle \mathcal{F}, \mathcal{R} \rangle$. The set of facts $\mathcal{F}$ contains all ground atoms in s, and $\mathcal{R}$ is defined as follows: For each action schema $A \in \mathcal{A}$ with parameters $\mathbf{X}$ and *pre*$(A) = \{Q_1, \ldots, Q_n\}$, $\mathcal{R}$ contains the *action applicability rule*

$$A\text{-applicable}(\mathbf{X}) \leftarrow Q_1, \ldots, Q_n,$$

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

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

Note that an action schema A produces $1 + |$*add*$(A)|$ rules. We assume all actions have at least one precondition and add a dummy precondition if this is not the case. Helmert (2009) showed that the canonical model $\mathcal{M}$ of $\mathcal{D}_s$ contains exactly the atoms that are reachable from s in Π^+.

In order to extract a heuristic value, Corrêa et al. (2021) assign a *weight* $w(r)$ to each rule r. Each action applicability rule is assigned a weight of $c(A)$ and all other weights are 0. They then consider a function v that is a maximal solution

for the following equations:

$$v(P) = \begin{cases} 0, & \text{if } P \in \mathcal{F} \\ v(f^{\text{add}}(P)), & \text{otherwise,} \end{cases} \quad (1)$$

$$v(r) = w(r) + \sum_{Q \in body(r)} v(Q) \quad (2)$$

$$f^{\text{add}}(P) \in \underset{\substack{r \in Ground(\mathcal{R}) \\ head(r)=P}}{\arg\min}\ v(r). \quad (3)$$

Equations (1) and (3) are defined over $\mathcal{M}$ and equation (2) is defined over $Ground(\mathcal{R})$. This system of equations has a unique maximal solution if all actions have costs larger than zero. In tasks with zero cost actions, these are considered to have a small cost of $\varepsilon > 0$ so the maximal solution is unique.

Corrêa et al. (2021) show that function v can be computed while also computing the canonical model. If the queue $\mathcal{Q}$ is ordered according to v, whenever an atom P is added to the canonical model, its v-value can be set according to (1). This order implicitly defines the achiever choice function f^{add} satisfying (3). The achiever $f^{\text{add}}(P)$ is called a *best achiever* of P. The authors also show that for a given state s, $v(goal) = h^{\text{add}}(s)$. Their algorithm uses an early stopping approach: when the value of $v(goal)$ is first computed, the algorithm can stop as $h^{\text{add}}(s)$ can already be extracted.

An important aspect is that their algorithm does not keep track of the ground rules explicitly: it adds labels to atoms when they are included in $\mathcal{M}$. An atom P is labeled with $\top$ if $P \in \mathcal{F}$, and with the body of its best achiever $body(f^{\text{add}}(P))$ otherwise. The best achiever $f^{\text{add}}(P)$ can then be reconstructed from the atom and its label.

The FF Heuristic

The h^{add} heuristic assumes that there is no positive synergy when achieving action preconditions. Thus h^{add} might overestimate the distance to the goal by a lot. To solve this issue, Hoffmann and Nebel (2001) introduce the FF heuristic, denoted as h^{FF}, which approximates h^+ better than h^{add}.

The original definition of h^{FF} (Hoffmann and Nebel 2001) does not specify an achiever choice function. While Hoffmann and Nebel choose the best achievers based on $h^{\max}$ values (Bonet and Geffner 2001), Keyder and Geffner (2008) do so using h^{add}. We follow Keyder and Geffner.

Let A_P be an action with $P \in add(A)$ that achieves P with minimum h^{add} cost. Let

$$\pi(P) = \begin{cases} \{\}, & \text{if } P \in s \\ \{A_P\} \cup \bigcup_{Q \in pre(A_P)} \pi(Q), & \text{otherwise.} \end{cases}$$

If we fix A_P for each reachable atom P and $goal$ is reachable, we can compute a unique solution to these equations by recursively evaluating it starting from $goal$. In this case, $\pi(goal)$ can be sequenced into a relaxed plan π_{FF} and the FF heuristic is defined as $h^{\text{FF}}(s) = c(\pi_{FF})$. If $goal$ is not reachable in the delete relaxation, then $h^{\text{FF}}(s) = \infty$.

We can compute h^{FF} from the canonical model of $\mathcal{D}_s$ by considering the derivation of $goal$ for the achiever choice function f^{add}. The set π_{FF} then matches the set of all A with parameters $\mathbf{X}$ for which A-applicable$(\mathbf{X})$ is part of the derivation.

Rule-Based FF

Corrêa et al. (2021) simplify $\mathcal{D}_s$ by removing the atoms A-applicable$(\mathbf{X})$. To do so, the action applicability rule A-applicable$(\mathbf{X}) \leftarrow Q_1, \ldots, Q_n$ is combined with the action effect rule $P \leftarrow A$-applicable$(\mathbf{X})$ into an *action rule* $P \leftarrow Q_1, \ldots, Q_n$ for each add effect P of A.

They also introduced further transformations that do not change the atoms in the canonical model but simplify the Datalog program. We will discuss those transformations in detail later on. For now we only consider their effect on an example. The transformations would transform this Datalog program

$$\begin{aligned} A\text{-applicable}(X,Y,Z) &\leftarrow Q_1(X), Q_2(X,Y), Q_3(Y,Z) \\ P_1(Y,Z) &\leftarrow A\text{-applicable}(X,Y,Z) \\ P_2(Y,Z) &\leftarrow A\text{-applicable}(X,Y,Z) \end{aligned}$$

into the following equivalent form:

$$\begin{aligned} P_1(Y,Z) &\leftarrow Aux(Y), Q_3(Y,Z) \\ P_2(Y,Z) &\leftarrow Aux(Y), Q_3(Y,Z) \\ Aux(Y) &\leftarrow Q_1(X), Q_2(X,Y). \end{aligned}$$

The first two rules do not mention X, so we cannot compute π_{FF} as before.

Corrêa et al. (2021) show that these transformations are performance-critical when computing h^{add}. We assume this also holds for h^{FF} and thus look into alternatives computing the heuristic value with the optimized Datalog program. As a first approximation, we consider a heuristic that just adds the weights of all rules in a derivation of $goal$. We call it the *rule-based FF heuristic* and denote it as $h^{\text{R-FF}}$.

Where h^{FF} computes a relaxed plan, $h^{\text{R-FF}}$ computes a set of ground rules that are sufficient to derive the goal in $\mathcal{D}_s$. The main difference between $h^{\text{R-FF}}$ and h^{FF} is that $h^{\text{R-FF}}$ might count the cost of the same action multiple times. This occurs when the action adds several atoms that are necessary to reach the goal. For example, in the Datalog program above, if both $P_1(Y,Z)$ and $P_2(Y,Z)$ are needed in the goal, the cost of action $A(X,Y,Z)$ is counted twice. This is similar to h^{add} but h^{add} counts the cost of an action every time one of its effects is used whereas $h^{\text{R-FF}}$ counts the cost of an action at most once for each of its effects. Our hypothesis is that actions usually add very few atoms that are necessary to reach the goal and thus the values of $h^{\text{R-FF}}$ and h^{FF} are close.

The intention of h^{FF} is to approximate h^+, so overcounting actions by treating their effects separately means that the heuristic loses accuracy. In the next section we introduce a general framework for associating a computation with a Datalog program. We will then show how h^{FF} can be computed in this framework without loss of accuracy.

Annotated Datalog

An *annotated Datalog program* is a Datalog program where every fact $P \in \mathcal{F}$ and every rule $r \in \mathcal{R}$ is annotated with a sequence of *instructions* denoted $ann(P)$ and $ann(r)$. An instruction can refer to the variables used in the rule. When a rule with instruction I is ground with substitution σ, we consider the ground rule to have instruction $I[\sigma]$.

Algorithm 1: Executing an annotated Datalog program.

```
1: function BACKCHAIN(P)
2:     if visited[P] then
3:         return
4:     visited[P] := True
5:     if P ∈ F then
6:         EXECUTE(ann(P))
7:     else
8:         for Q ∈ BODY(f(P)) do
9:             BACKCHAIN(Q)
10:        EXECUTE(ann(f(P)))
```

The semantics of an annotated Datalog program are relative to the derivation of an atom P. Recall that a derivation is a sequence over $\mathcal{F} \cup \mathit{Ground}(\mathcal{R})$. To *execute* a Datalog program for the derivation of P, we map each element of this sequence to its associated instruction and execute the instructions in this order.

Algorithm 1 shows how to find a derivation of an atom and execute the Datalog program for it. The back-chaining procedure handles each atom at most once (lines 2–4). Facts in $\mathcal{F}$ need no further derivation, so their annotation is executed directly (lines 5–6). For other atoms, the algorithm ensures that their derivation is included and the corresponding instructions are executed before executing the instructions of the achiever (lines 8–10).

We demonstrate that annotated Datalog programs are useful by expressing different concepts in them. We base all examples on the Datalog program $\mathcal{D}_s$ and the achiever choice function f^{add}, as defined in (3). Other achiever choice functions, such as the one based on $h^{\max}$, are also possible.

Useful Atoms An atom is considered *useful* (Hoffmann and Nebel 2001) if it is not in the state but required in the goal or the precondition of an operator used to reach another useful atom. In $\mathcal{D}_s$, they are the atoms occurring in a derivation of the goal except for $\mathcal{F}$ and atoms of the form A-applicable$(\mathbf{X})$ (Corrêa et al. 2021). To compute them with annotated Datalog programs, we use the annotation

$$\mathit{ann}(r) = [\texttt{Mark } \mathit{head}(r) \texttt{ as useful}]$$

for all action effect rules $r \in \mathcal{R}$ and an empty annotation for all action applicability rules and all $P \in \mathcal{F}$.

Rule-Based FF The heuristic $h^{\text{R-FF}}$ can also be expressed with an annotated Datalog program. For each action A, we annotate its corresponding action effect rules r with

$$\mathit{ann}(r) = [\texttt{Add } c(A) \texttt{ to h}]$$

and use an empty annotation in all other cases. The variable `h` contains the value of $h^{\text{R-FF}}(s)$ after the execution.

FF For all actions A we annotate the corresponding action applicability rule $r = A\text{-applicable}(\mathbf{X}) \leftarrow Q_1, \ldots, Q_n$ with

$$\mathit{ann}(r) = [\texttt{Include } A(\mathbf{X}) \texttt{ in } \pi_{\text{FF}}]$$

and use empty annotations in all other cases. When executing an instruction for an atom P, if $P \notin s$, the instruction adds the achiever $A(\mathbf{X}) = A_P$ to π_{FF}. We can show by structural induction that after the execution for P, the variable π_{FF} contains all actions of $\pi(P)$. The base case of $P \in s$ is trivial ($\pi(P) = \emptyset$). In the inductive step, the achiever choice of P is $A = f^{\text{add}}(P)$ which is added to π_{FF} by the annotation of the rule achieving P. The actions required to achieve preconditions of this action are already included in π_{FF} according to the induction hypothesis. As the derivation of P only relies on actions in $\pi(P)$, no additional actions are included in the set, so π_{FF} has the value of $\pi(\mathit{goal})$ after the execution for goal.

We introduced rule-based FF specifically because the optimizations introduced by Corrêa et al. (2021) are important for performance. Yet, all examples above use the unoptimized program $\mathcal{D}_s$. To make use of the optimizations, we will now discuss how such optimizations can be done on any annotated Datalog program without changing its semantics.

Transformations of Annotated Datalog

We introduce four transformations of annotated Datalog programs: *rule merging*, *rule splitting*, *predicate collapsing*, and *variable renaming*. These transformations change the set of rules and predicate symbols used in the program which affects the derivations and thus the semantics of the program. However, we can still show that the semantics before and after the transformation are equivalent in the following sense: For any derivation under an achiever choice function f in the transformed program, there is a derivation under an achiever choice function f' in the original program such that the execution of both problems under these derivations produces the same results. Moreover, in our use case, all atoms of the planning task that have a certain v-value under f have the same v-value under f'. This implies that any execution based on h^{add} achievers in the transformed program corresponds to an execution in the original program for one of the possible choices of h^{add} achievers. The non-determinism is completely caused by the tie-breaking in h^{add}-achievers.

The transformations we introduce are very similar to the optimizations used by Corrêa et al. (2021). However, their optimizations must be applied in a specific order, while our transformations do not depend on the history of previous transformations. Additionally, we have to update the annotations, whereas their Datalog programs do not have them.

Rule Merging Rules can be merged to get rid of intermediate atoms such as A-applicable$(\mathbf{X})$ in $\mathcal{D}_s$. For an atom P let $\mathcal{R}_P^+ \subseteq \mathcal{R}$ be the rules with head P and let $\mathcal{R}_P^- \subseteq \mathcal{R}$ be the rules where the body contains P. The rules can only be merged if those sets do not overlap, there are no other rules or facts with the predicate symbol of P, and rules from $\mathcal{R}_P^+$ do not share variables with rules from $\mathcal{R}_P^-$ other than the ones mentioned in P. We can then replace $\mathcal{R}$ by $(\mathcal{R} \setminus (\mathcal{R}_P^+ \cup \mathcal{R}_P^-)) \cup \mathcal{R}'$. The set of merged rules $\mathcal{R}'$ contains the rule $r^{\pm} = \mathit{head}(r^-) \leftarrow (\mathit{body}(r^-) \setminus \{P\}) \cup \mathit{body}(r^+)$ for every combination of rules $r^+ \in \mathcal{R}_P^+$ and $r^- \in \mathcal{R}_P^-$. The weight of the merged rule is $w(r^{\pm}) = w(r^+) + w(r^-)$ and its annotation is $\mathit{ann}(r^{\pm}) = [\mathit{ann}(r^+)\texttt{;}\ \mathit{ann}(r^-)]$.

For example, the rules from our previous example

$$r_1 = A\text{-applicable}(X,Y,Z) \leftarrow Q_1(X), Q_2(X,Y), Q_3(Y,Z)$$
$$r_2 = P_1(Y,Z) \leftarrow A\text{-applicable}(X,Y,Z)$$
$$r_3 = P_2(Y,Z) \leftarrow A\text{-applicable}(X,Y,Z)$$

can be replaced by the rules

$$r_{1,2} = P_1(Y,Z) \leftarrow Q_1(X), Q_2(X,Y), Q_3(Y,Z)$$
$$r_{1,3} = P_2(Y,Z) \leftarrow Q_1(X), Q_2(X,Y), Q_3(Y,Z).$$

Assume we want to compute h^{FF} and useful atoms, then $ann(r_1)$ would be $[\texttt{Include } A(\mathbf{X}) \texttt{ in } \pi_{\text{FF}}]$ and $ann(r_i) = [\texttt{Mark } head(r_i) \texttt{ as useful}]$ for $i \in \{2,3\}$. In that case, we would have

$$ann(r_{1,i}) = [\texttt{Include } A(\mathbf{X}) \texttt{ in } \pi_{\text{FF}}\texttt{;}\ \texttt{Mark } head(r_i) \texttt{ as useful}].$$

A derivation that uses a ground rule $r^- \in Ground(\mathcal{R}_P^-)$ must contain a ground rule $f(P) \in Ground(\mathcal{R}_P^+)$ to derive P. Those two rules can be replaced by their corresponding grounded merged rule $r^{\pm}$ in the transformed program. Likewise a merged rule in a derivation for the transformed program can be replaced by its components in the original program. The derived value, its v-value, and the sequence of executed instructions is the same.

Rule Splitting The *rule-splitting* transformation divides rules with large bodies into multiple smaller rules. The goal is to create an implicit join tree for the rules of the Datalog program (Helmert 2009). Let $r = P \leftarrow Q_1, \ldots, Q_n$ be a rule in an annotated Datalog program and let $1 \leq k \leq n$. Let $\mathbf{X}$ be the set of variables defined as

$$\mathbf{X} = \left(\bigcup_{0 \leq i \leq k} vars(Q_i) \right) \cap \left(\bigcup_{k+1 \leq i \leq n} vars(Q_i) \cup P \right).$$

By introducing a new predicate symbol Aux, rule r can be split into two rules $r_1 = Aux(\mathbf{X}) \leftarrow Q_{k+1}, \ldots, Q_n$ with $w(r_1) = 0$, and $r_2 = P \leftarrow Q_1, \ldots, Q_k, Aux(\mathbf{X})$ with $w(r_2) = w(r)$. The annotation $ann(r_1)$ stores the values of the variables $\bigcup_{k+1 \leq i \leq n} vars(Q_i)$ that r_1 depends on. The annotation $ann(r_2)$ is the same as $ann(r)$ but replacing those variables by the values stored by $ann(r_1)$.

Rule $r_{1,2} = P_1(Y,Z) \leftarrow Q_1(X), Q_2(X,Y), Q_3(Y,Z)$ from the previous example could be split into

$$r_1 = Aux(Y) \leftarrow Q_1(X), Q_2(X,Y),$$
$$r_2 = P_1(Y,Z) \leftarrow Aux(Y), Q_3(Y,Z)$$

If $r_{1,2}$ had the annotation $[\texttt{Add } A(X,Y,Z) \texttt{ to } \pi_{\text{FF}}]$ the new rules would have the annotations

$$ann(r_1) = [\texttt{Instantiation}[Aux(Y)] = X,Y]$$
$$ann(r_2) = [\texttt{X,Y} = \texttt{Instantiation}[Aux(Y)];\ \texttt{Add } A(\texttt{X,Y},Z) \texttt{ to } \pi_{\text{FF}}].$$

Executing $ann(r)$ has the same effect as executing $ann(r_2)$ after $ann(r_1)$. As Aux never occurs outside of r_1 and r_2, any achiever choice function for $Aux(\mathbf{X})$ must map to r_1 ground with $\mathbf{X}$. In a derivation, this rule will therefore occur before r_2 leading to the execution of $ann(r_1)$ before $ann(r_2)$. So, rule splitting does not change the semantics of the annotated Datalog program.

Rule splitting might reduce the computational effort to construct the canonical model $\mathcal{M}$. Assume $Q_1(a)$ was just popped from the queue and we have to find a rule $r \in Ground(r_{1,2})$ such that $\mathcal{M} \models body(r)$. If we first join $Q_1(X)$ with $Q_3(Y,Z)$ we could get a larger intermediate result than if we first join with $Q_2(X,Y)$. In the split rules, only the efficient join is possible.

Predicate Collapsing When the atoms of two predicate symbols P_1 and P_2 are reachable in the same ways throughout the Datalog program, we know that if $P_1(\mathbf{X})$ has a certain derivation, then $P_2(\mathbf{X})$ can have the same derivation. In that case, we can replace $P_1(\mathbf{X})$ with $P_2(\mathbf{X})$ without changing the semantics of the Datalog program. Before we express this formally, consider a Datalog program with following rules as an example:

$$r_1 = P_1(X) \leftarrow Q(X,Y), R(Y),$$
$$r_2 = P_2(X) \leftarrow Q(X,Y), R(Y),$$
$$r_3 = R(Z) \leftarrow Q(Z,Y), P_1(Y),$$
$$r_4 = R(Z) \leftarrow S(Z,Y), P_2(Y)$$

where the weights and annotations of r_1 and $_2$ are identical. In this example, we can replace P_1 by P_2 without affecting the semantics. Note that since the rules of a Datalog program are a set, r_1 and r_2 become identical, so the resulting Datalog program only contains three rules. Reducing the number of predicate symbols also reduces the size of $\mathcal{M}$, which might speed up the computation of the heuristics.

Formally, two predicates P_1 and P_2 can be collapsed if for each rule $r_1 : P_1(\mathbf{X}) \leftarrow Q_1, \ldots, Q_n$, there is a rule $r_2 : P_2(\mathbf{X}) \leftarrow Q_1, \ldots, Q_n$, with $ann(r_1) = ann(r_2)$ and $w(r_1) = w(r_2)$. For this purpose, we treat facts $A \in \mathcal{F}$ as rules with an empty body, i.e. $A \leftarrow \top$. If we then replace P_1 by P_2, any derivation of $goal$ in the resulting Datalog program corresponds to a derivation in the original program and the v-values of all derived facts and rules remain the same.

Variable Renaming Variable names used in a rule r have no impact on the canonical model or the back-chaining through a derivation. They can be renamed without changing the semantics of a Datalog program as long as the variables occurring in $ann(r)$ are renamed accordingly.

Transformations Used in Our Experiments While the transformations are more general, we limit their use to simulate the optimizations done by Corrêa et al. (2021) for better comparability. We start by merging action applicability and action effect rules. We then use the strategy of Corrêa et al. to split rules into rules with binary bodies. In all resulting rules, we rename the variables to canonical names to maximize the number of predicate symbols the algorithm can collapse. We finally collapse auxiliary predicate symbols where possible.

Transformations		Action applicability rules	
		not merged	merged
Auxiliary Predicates	not collapsed	1072	1144
	not collapsed + VR	1069	1139
	collapsed	1088	1176
	collapsed + VR	**1100**	**1244**

Table 1: Coverage of h^{FF} with eager GBFS on both benchmark sets (1863 tasks) under different Datalog transformations. All runs include the rule-splitting transformation.

Experimental Results

We implemented the Datalog-based FF heuristic using the Powerlifted (PWL) planner by Corrêa et al. (2020). The source code of our implementation is available online (Corrêa et al. 2022). Our experiments were run on an Intel Xeon Silver 4114 processor running at 2.2 GHz using a run time limit of 30 minutes and a memory limit of 16 GiB per task.

We benchmark our algorithms on two sets. The first set contains 1001 IPC tasks from 29 STRIPS domains. This is the same set used by Corrêa et al. (2021). The second set contains 862 hard-to-ground (HTG) tasks over 11 different domains. The HTG set is the same used by Lauer et al. (2021). We always consider action schemas as unit-cost.

We tested the following heuristics: *(i)* the lifted $h^{\text{R-FF}}$ heuristic and the h^{FF} heuristic; *(ii)* the lifted h^{add} heuristic (Corrêa et al. 2021); *(iii)* the lifted goal-count heuristic, denoted as h^{gc} (Corrêa et al. 2020); *(iv)* the lifted h^{gc} heuristic using the unary relaxation heuristic as a tie-breaker, denoted as $h^{\text{gc, ur}}$, and its enhanced version using disambiguation of static predicates, $h^{\text{gc, ur-d}}$(Lauer et al. 2021). These are the best two configurations in the experimental results by Lauer et al.; and *(v)* the ground version of h^{FF} from Fast Downward (FD), release 20.06 (Helmert 2006).

We tested h^{add} and h^{FF} (both lifted and ground versions) using a simple Eager Greedy Best-First Search (Pohl 1970), and also using the Lazy GBFS with preferred operators and a boosted dual-queue (Richter and Helmert 2009; Corrêa et al. 2021).[1] We write these simply as "Eager" and "Lazy + PO".

Transformations Corrêa et al. (2021) observed that all transformations of the Datalog program are beneficial for h^{add}. To test if this occurs with h^{FF}, we ran all combinations of rule merging, predicate collapsing and variable renaming (VR). The algorithm by Helmert (2009) used to compute the canonical model requires all rules in a specific form ensured by rule splitting. Thus, we cannot disable rule splitting.

Table 1 shows the coverage of a GBFS with h^{FF} after different transformations. The results confirm that the finding by Corrêa et al. (2021) also holds for h^{FF}: all transformations are beneficial and in particular removing action predicates by merging rules is critical for performance. Using all transformations achieves the best performance, increasing the baseline coverage from 1072 to 1244.

Removing action predicates always increases the coverage by at least 70 tasks. However, merging rules can affect how ties in the achiever choice function are broken which affects the performance. The benefit of merged rules is thus not a clear dominance and we saw a few IPC domains where coverage decreased (e.g., pipesworld-split, trucks). The variable renaming transformation is mainly useful in conjunction with other transformations as they create more rules that become identical with canonical variable names.

Datalog-Based Heuristics Our next experiment compares the performance of different Datalog-based heuristics: h^{add}, $h^{\text{R-FF}}$, and h^{FF}. The first seven columns of Table 2 show the coverage for these heuristics. We include detailed results for the IPC set only on Zenodo (Corrêa et al. 2022) due to space limitation. With both eager and lazy searches, coverage improves when switching from h^{add} to h^{FF}, and we see a larger improvement switching from Eager (which does not use preferred operators) to Lazy + PO. These confirm similar results in ground planning (Richter and Helmert 2009) showing that while h^{FF} is generally an improvement over h^{add}, using preferred operators impacts the search performance more. As with ground planning, the results for different domains vary and h^{add} sometimes gives better guidance than h^{FF} (e.g., h^{add} solves 5 tasks more in satellite). As h^{add} is more greedy than the other methods, this is expected in tasks where greedy behavior leads to a plan more quickly.

With eager search, the lifted h^{FF} has an edge over the simpler rule-based FF heuristic $h^{\text{R-FF}}$, although $h^{\text{R-FF}}$ has superior coverage in a few domains (e.g., airport). However, using the lazy search, h^{FF} and $h^{\text{R-FF}}$ have a similar performance: h^{FF} solves only 13 instances more than $h^{\text{R-FF}}$ in total. In this case, $h^{\text{R-FF}}$ solves more tasks than h^{FF} in 5 domains but h^{FF}solves more tasks in 7 domains with larger differences in coverage (e.g., +11 tasks in openstacks).

Compared to h^{add} the overestimation done by $h^{\text{R-FF}}$ is limited by the maximal number of effects in an action. We hypothesized that usually not all effects of an action are required and thus the overestimation would be low. To test this, we measured the number of useful atoms per action in the relaxed plan found by $h^{\text{R-FF}}$ in the initial state. If this proportion p is 1 then $h^{\text{R-FF}}$ is equal to h^{FF} for this state. With higher values, the overestimation is larger. Among the 1863 tasks, 87.3% (1627) have $p \leq 5$, while only 10.5% (196) have $p \leq 2$. This shows that while the overestimation of $h^{\text{R-FF}}$ is still low, it is not as low as we initially expected. Also note that this overestimation does not necessarily influence the heuristic quality as scaling all heuristic values with the same factor has no effect in our search algorithms.

Other Lifted Planners We also investigated how our methods compare to the other lifted heuristics in the literature that do not use Datalog. We ran experiments using h^{gc} (Corrêa et al. 2020), and with the heuristics $h^{\text{gc, ur}}$ and $h^{\text{gc, ur-d}}$ (Lauer et al. 2021) that break ties in h^{gc} based on the unary relaxation of the delete-relaxed task. Because of this further relaxation, it is currently not possible to extract a relaxed plan or preferred operators from these estimates.

[1] Note that the notion of preferred operators is not the same for each case, as explained by Corrêa et al. (2021).

	Datalog-Based Heuristics						Other Lifted Methods			Fast Downward	
	Eager			Lazy + PO			Eager			Eager	Lazy + PO
Coverage	h^{add}	h^{FF}	$h^{\text{R-FF}}$	h^{add}	h^{FF}	$h^{\text{R-FF}}$	h^{gc}	$h^{\text{gc, ur-d}}$	$h^{\text{gc, ur}}$	h^{FF}	h^{FF}
IPC Sum (1001)	629	702	677	759	820	816	597	575	569	775	**862**
blocksworld-large (40)	1	4	0	6	9	4	4	7	7	4	**12**
childsnacks-large (144)	34	27	30	82	73	69	26	98	65	51	**115**
genome-edit-distance (312)	185	294	225	289	311	310	**312**	**312**	**312**	**312**	**312**
logistics-large (40)	8	9	9	**40**	**40**	**40**	20	0	0	30	32
organic-synthesis (56)	47	48	48	**49**	48	**49**	48	47	47	20	20
pipesworld-tankage-nosplit (50)	22	23	25	28	**32**	**32**	22	10	12	15	19
rovers-large (40)	11	36	36	**40**	**40**	**40**	1	16	14	11	13
visitall-multidimensional (180)	118	101	104	143	143	143	65	**151**	102	72	72
HTG Sum (862)	426	542	477	677	**696**	687	498	641	559	515	595
Total Sum (1863)	1055	1244	1154	1436	**1516**	1503	1095	1216	1128	1290	1457

Table 2: Coverage for all tested methods over both benchmark sets. Best results shown in bold typeface.

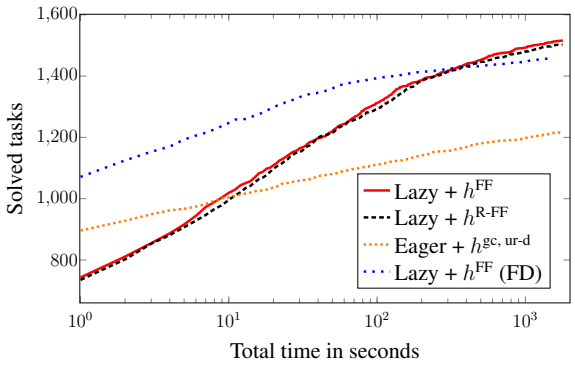

Figure 1: Solved tasks over time for different methods.

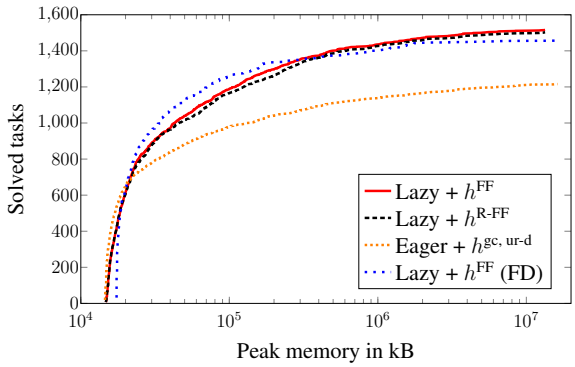

Figure 2: Solved tasks for different memory limits.

We thus use these heuristics with Eager. The columns under "Other Lifted Methods" in Table 2 show the results.

When focusing on Eager search, all three methods based on h^{gc} are competitive with or better than h^{FF} and $h^{\text{R-FF}}$. In total h^{FF} solves 28 tasks more than $h^{\text{gc, ur-d}}$. The advantage is mainly in the IPC domains where h^{FF} solves 127 tasks more than $h^{\text{gc, ur-d}}$. On the HTG domains, the picture is reversed and h^{FF} solves 99 tasks less than $h^{\text{gc, ur-d}}$.

All methods using Lazy + PO are superior in coverage. All the Datalog-based heuristics are still better on the IPC set but also solve more tasks on the HTG set. As mentioned above, the preferred operators have a larger impact than a better heuristic and it is not known how to find preferred operators over the unary relaxation task efficiently.

We analyzed coverage over time for h^{FF}, $h^{\text{R-FF}}$, and $h^{\text{gc, ur-d}}$ in Figure 1. The search using $h^{\text{gc, ur-d}}$ has much higher coverage in approximately the first 10 seconds because it is very fast to compute. Hence, tasks that do not require a deep search are solved quickly. In fact, the lifted h^{FF} and $h^{\text{R-FF}}$ computation are worst-case exponential in the PDDL-size, while $h^{\text{gc, ur-d}}$ is polynomial. Still, for small tasks the overhead of computing h^{FF} or $h^{\text{R-FF}}$ does not pay off. For larger tasks, the stronger heuristic guidance is worth spending more time to compute h^{FF} and $h^{\text{R-FF}}$.

When analyzing memory, our methods are superior to $h^{\text{gc, ur-d}}$ even with small limits. Figure 2 compares coverage for different limits on peak memory. Using a limit as low as 100 MiB, the difference in coverage between our methods and $h^{\text{gc, ur-d}}$ is already larger than 150 tasks.

Ground Planners Our last experiment compares PWL with the lifted h^{FF} heuristic FD (Helmert 2006) with the ground h^{FF} heuristic. Both use a generalized Dijkstra algorithm for their heuristic computation but the lifted implementation requires a more expensive unification step with this algorithm. Additionally, the two planners break ties in the heuristic computation differently and have a different notion of preferred operators (Corrêa et al. 2021), so they are not guaranteed to expand the same set of states. We thus treat this experiment as comparing two planners rather than comparing two implementations of h^{FF}.

Using Eager, FD outperforms PWL in general. In the IPC set, it solves 73 more tasks in total. In 9 domains, FD solved 5 or more tasks more than PWL. The largest difference was in barman, where FD solved 11 tasks more. The only do-

mains where PWL solves more tasks are visitall, parking-sat14, and thoughtful. This difference in performance is expected since the IPC tasks are easy to ground and the main challenge is the search itself. In such cases the advantage of not having to ground the task does not offset the more expensive heuristic computation in PWL. When comparing the results in the HTG set, the planners are on par. Nonetheless, the coverage in some domains differs a lot, such as in logistics-large and organic-synthesis.

With Lazy + PO, the trend is similar: FD is superior on the IPC set, while PWL is superior on the HTG set. However, on the IPC set, the advantage of FD reduces from 73 to 42 tasks. On the HTG set, the advantage of PWL increases from 27 to 101 tasks. In total, PWL has the highest coverage in this setting.

As Figure 1 shows, FD solves more tasks than the lifted methods in the first seconds. As most of the tasks in the combined benchmark are easy-to-ground and the ground heuristic computation is much faster, FD solves even more tasks than $h^{\text{gc, ur-d}}$ straight away. In fact, PWL with h^{FF} only passes FD in number of solved tasks after around 400 seconds.

Comparing memory, FD has a similar performance to our methods. We can see in Figure 2 that with limits smaller than 1 GiB, FD has the highest coverage. For larger limits, the lifted planner is slightly superior.

Conclusion

We showed how to compute h^{FF} from a lifted representation of a planning task. The rule-based FF heuristic, $h^{\text{R-FF}}$, treats the effects of an action independently and can thus overcount an action. Therefore, we introduced annotated Datalog programs, which associate every rule and fact with an instruction. After evaluating a Datalog query, the actions belonging to a relaxed plan can be extracted by executing the annotations along its derivation. We showed how such annotated Datalog programs can be simplified without changing their semantics. This allows us to compute h^{FF} from smaller annotated Datalog programs which are faster to evaluate. We believe annotated Datalog programs can be used beyond the definition of heuristics, such as for generating landmarks (Zhu and Givan 2003), as they allow us to express other computations over the relaxed task.

Empirically, a lifted planner using the lifted FF heuristic achieves state-of-the-art performance among lifted methods. In domains that are hard to ground, the lifted planner solves more tasks than any other method. In IPC domains, which are not particularly hard to ground, it reduces the gap to Fast Downward, a state-of-the-art ground planner, and achieves better coverage in a few of them.

Acknowledgments

This work was funded by the Swiss National Science Foundation (SNSF) as part of the project "Certified Correctness and Guaranteed Performance for Domain-Independent Planning" (CCGP-Plan). Furthermore, this research was also partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under grant agreement no. 952215.

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Betz, C.; and Helmert, M. 2009. Planning with h^+ in Theory and Practice. In *ICAPS 2009 Workshop on Heuristics for Domain-Independent Planning*, 64–69.

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *AIJ*, 129(1): 5–33.

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

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

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2022. Code from the AAAI 2022 paper "The FF Heuristic for Lifted Classical Planning". https://doi.org/10.5281/zenodo.6373793. Accessed: 2022-04-11.

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

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

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

Hoffmann, J. 2005. Where 'Ignoring Delete Lists' Works: Local Search Topology in Planning Benchmarks. *JAIR*, 24: 685–758.

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

Horčík, R.; and Fišer, D. 2021. Endomorphisms of Lifted Planning Problems. In *Proc. ICAPS 2021*, 174–183.

Keyder, E.; and Geffner, H. 2008. Heuristics for Planning with Action Costs Revisited. In *Proc. ECAI 2008*, 588–592.

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

McDermott, D. 1996. A Heuristic Estimator for Means-Ends Analysis in Planning. In *Proc. AIPS 1996*, 142–149.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine*, 21(2): 35–55.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *AIJ*, 1: 193–204.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Richter, S.; and Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. In *Proc. ICAPS 2009*, 273–280.

Zhu, L.; and Givan, R. 2003. Landmark Extraction via Planning Graph Propagation. In *ICAPS 2003 Doctoral Consortium*, 156–160.