Unary Relaxation

BSc Thesis

Pascal Lauer

Saarland University
Foundations of Artificial Intelligence Group
Saarland Informatics Campus

July 2020

Acknowledgment

Foremost, I would like to thank Jörg Hoffmann for giving me the opportunity to write this thesis, as well as Álvaro Torralba, Daniel Höller and Julia Wichlacz for the constant support with this thesis. I would not know how this should have been possible without you.

I would also like to thank Augusto B. Corrêa for the contious and really quick support for any issues within the Power Lifted Planner.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 27 July 2020 (Pascal Lauer)

Abstract

Most techniques of domain independent planning are defined within a grounded setting, e.g. FDR or STRIPS. Still, it is typical to define domains in PDDL, a lifted representation. Recent research has shown, that in some domains, the transformation from the lifted representation into the grounded equivalent comes with a undesired space blow up. Which can in some cases be big enough, that the problem can't be represented on standard machines anymore. Recent work introduced a lifted successor generation, what can be intuitively understood as applying this transformation (called grounding) on demand.

In this thesis I will extend this approach. I will first introduce a new relaxation, called unary relaxation. Within this relaxation I construct a heuristic that is able operate within the space bounds of lifted representation, without having to sacrifice run-time performance. I analyze the performance of the heuristic and introduce extensions that address the identified bottlenecks of the heuristic.

Contents

Chapter 1

Introduction

AI planning is a research area that revolves around the question, how to solve decision making problems. The setting of this paper is within domain independent planning, where given an arbitrary decision making problem, a planner should find a sequence of decisions that will lead to the goal. It is typical to create models ,that describe tasks, to address arbitrary problems. These models contain facts (i.e. the things that can be achieved), an initial state (i.e. what is already achieved in the beginning), a goal (i.e. what shall be achieved) and the decisions that can be made to achieve new facts (potentially at the cost of losing an previously achieved fact and just under a certain precondition). Years ago, heuristic search was discovered to be promising. **??** By now it is one of the most, if not the most, prominent approach within planning. In heuristic search the planer builds a graph-like structure, called state space, from the model. In this structure nodes are states (i.e. sets of facts) and the edges are described by the decisions, here called actions. Heuristics are used to guide a search within this structure and potentially achieve the goal faster.

So far most heuristics and search algorithms in domain independent planning are defined within a grounded setting. FDR, also known as SAS^+ [2], and STRIPS [4] are the most common ways for such a grounded setting. However, in practice, domains (e.g. all domains of the International Planning Competition (IPC)) are typically defined in a lifted representation, called PDDL. [12] PDDL allows to represent domains in a really compact, but readable, way. In order to search within such a domain, the lifted representation will be grounded, i.e. transferred into a grounded representation. But this comes at a cost. Even if there have been optimizations [6], for the grounding process over the years, losing the compactness of the lifted presentation has shown to be undesirable in some cases. A prime example for this is the Organic Synthesis domain. [10] Recent work has shown that computing the model of full domains in Fast Downward [8] can be as high as 128GB. [11] Using a model like this still seems infeasible for most machines as of today and is also far fetched from the 8GB memory limit that was used in the latest IPC.

For such domains it seems desirable to have algorithms that can work directly on the lifted representation, without coming at the cost of a huge run time blow up. Recent work introduced a planning framework that supports a lifted successor generation, i.e. the framework is able to search the state space without grounding the representation. However, the framework does not provide any heuristics but goal grounding, which is a really basic heuristic. A really successful heuristic within a grounded setting is h^{FF}. **??** It creates a relaxed task, meaning a task that is similar to the original one, but easier to solve. In particular the relaxation is called delete relaxation. What this means will be explained in the background section. h^{FF} then extracts a plan in the relaxed model and uses the plan cost as heuristic estimate. I introduce a new relaxation, called Unary Relaxation, which is an extension to the the delete relaxation. It enables computing heuristic values within polynomial time, with respect to the lifted representation, in a h^{FF} fashion.

The additional relaxation comes with a loss of information that leads to bad heuristics estimates on a lot of domains. I will inspect some pitfalls and extend my approach that address them. Furthermore I will evaluate the refined approach and list still existing issues. With that I want to give some context of limitations and opportunities of my approach and lifted planning in general.

Chapter 2

Background

Here I will formalize the principles of planning and give the necessary background. I will first introduce the running example to explain all definitions with the help of it.

2.1 Running Example

As running example I will use the IPC domain Transport. The domain is inspired by a real world logistics task where trucks need to deliver packages to certain destinations. In the beginning each truck is placed at some location and each package is placed at some location or into some truck. To solve the task, trucks can perform the following actions:

- A truck can travel from location a A to a location B iff A and B are connected.
- A truck can load packages at its current location.
- A truck can unload previously loaded packages to its current location.
- Trucks can load unlimited packages.

The goal is reached when all packages are delivered.

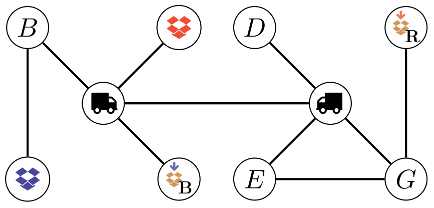


Figure 2.1: A simple illustration of a transport task.

Figure 2.1 illustrates an example of a Transport task. There are two trucks. I will refer to the left truck as T_L and to the right truck as T_R. I will refer to their starting locations as S_{T_L} and S_{T_R} respectively. There are also two packages. The R(ed) and B(lue) package. I will refer to the packages as P_R and P_B respectively. Their starting location is marked by the respectively colored package and will be referred to as S_{P_R} and S_{P_B}.

Their destination is marked by an arrow above the package and is annotated with their respective name. I will refer to the destinations as D_{P_R} and D_{P_B} respectively. The set $\mathit{Locations} := \{S_{P_B}, S_{P_R}, S_{T_L}, S_{T_R},, D_{P_B}, D_{P_R}, B, D, E, G\}$ denotes all locations within the task.

2.2 Lifted and Grounded Planning

I will start by defining a grounded task that corresponds to a typical STRIPS representation. [4]

Definition 1: (Grounded Task)
A Grounded Task is a tuple $(\mathcal{F}, \mathcal{A}_G, \mathcal{I}_G, \mathcal{G}_G)$ where $\mathcal{F}$ is a set of facts, $\mathcal{A}$ is a set of grounded actions, $\mathcal{I}_G \subseteq \mathcal{F}$ is the grounded initial state, $\mathcal{G}_G \subseteq \mathcal{F}$ is the grounded goal.

- An action $a \in \mathcal{A}_G$ is a triple of fact sets (pre_a, add_a, del_a) with $add_a \cap del_a = \emptyset$.
- A state $s \subseteq \mathcal{F}$ is a set of facts.
- An action $a \in \mathcal{A}_G$ is called applicable in s if $pre_a \subseteq s$.
- Applying a to s (denoted by $s[\![a]\!]$) results in a state $s' = (s \setminus del_a) \cup add_a$. s' is called a successor of s.
- Applying a sequence of actions $a_1, ..., a_n$ to a state s is defined as $s[\![a_1, ..., a_n]\!] := s[\![a_1]\!][\![a_2, ..., a_n]\!]$.
- A sequence of actions $a_1, ..., a_n$ is called a plan, if $\mathcal{G}_G \subseteq \mathcal{I}_G[\![a_1, ..., a_n]\!]$.

Example 1: (Grounded Transport Task)
The task from figure 2.1 can be formalized as grounded task in the following way:

- Intuitively speaking facts are the things that differentiate states from another. For the Transport domain these things are the location of the truck and the location of the packages. So the set of facts is:

$$\begin{aligned}\mathcal{F} := &\{at(T, L) \mid T \in \{T_1, T_2\}, L \in \mathit{Locations}\}\\ \cup\ &\{at(T, L) \mid P \in \{P_R, P_B\}, L \in \mathit{Locations}\}\\ \cup\ &\{in(P, T) \mid P \in \{P_R, P_B\}, T \in \{T_1, T_2\}\}\end{aligned}$$

- Facts change when a trucks drives to another location, loads a package or unloads a package. Thus the actions can be defined as follows:

$$\begin{aligned}\mathit{DriveActions} := \{(&\{at(T, L_1)\}, \{at(T, L_2)\}, \{at(T, L_1)\})\\ &\mid T \in \{T_L, T_R\}; L_1, L_2 \in \mathit{Locations};\\ &\quad L_1 \text{ and } L_2 \text{ are connected in the picture}\}\end{aligned}$$

$$\begin{aligned} \mathit{LoadActions} \coloneqq \{(\{&\mathit{at}(T,L), \mathit{at}(P,L)\}, \{\mathit{at}(P,L)\}, \{\mathit{in}(P,T)\}) \\ &\mid T \in \{T_L, T_R\}, L \in \mathit{Locations}, P \in \{P_1, P_2\}\} \end{aligned}$$

$$\begin{aligned} \mathit{UnloadActions} \coloneqq \{(\{&\mathit{at}(T,L), \mathit{in}(P,T)\}, \{\mathit{in}(P,T)\}, \{\mathit{at}(P,L)\}) \\ &\mid T \in \{T_L, T_R\}, L \in \mathit{Locations}, P \in \{P_1, P_2\}\} \end{aligned}$$

$$\mathcal{A} \coloneqq \mathit{DriveActions} \cup \mathit{LoadActions} \cup \mathit{UnloadActions}$$

- The initial state is the set of facts that are initially true, this means $\{\mathit{at}(T_L, S_{T_L}), \mathit{at}(T_R, S_{T_R}), \mathit{at}(P_B, S_{P_B}), \mathit{at}(P_R, S_{P_R})\}$.
- The goal is reached when both packages reached their destination. Meaning the goal is $\{\mathit{at}(P_B, D_{P_B}), \mathit{at}(P_R, D_{P_R})\}$
- Here I will refer to an action in DriveActions as $\mathit{drive}(t, l_1, l_2)$, where t, l_1 and l_2 are an according valid choice for T, L_1 and L_2 in the set builder condition of DriveActions. $\mathit{load}(t,p,l)$ and $\mathit{unload}(t,p,l)$ are defined in the same way. Then the following sequence of actions could be used to deliver the blue package:

  $$\begin{aligned} &\mathit{drive}(T_L, S_{T_L}, B),\ \mathit{drive}(T_L, B, S_{P_B}),\ \mathit{load}(T_L, P_B, S_{P_B}),\ \mathit{drive}(T_L, S_{P_B}, B), \\ &\qquad \mathit{drive}(T_L, B, S_{T_L}), \mathit{drive}(T_L, S_{T_L}, D_{P_B}), \mathit{unload}(T_L, P_B, D_{P_B}) \end{aligned}$$

  Extending this action sequence s.t. the red package will be also delivered (and the blue package did not change its position) yields a plan. Note that it is not mandatory to use both trucks.

In this example of our task, the grounded representation seems really compact, if you judge it by the amount of text I have written down. But realize that this is just because I used the set-builder notation. Representing these sets within a data structure means that each fact will be listed separately.

Still, the set-builder notation is really handy to represent planning tasks. And this is why PDDL is designed to serve the same purpose. This means the lifted representation will be defined in a way that actions and (the equivalent for) facts can be represented in such a compact, parameterized way. Instead of directly representing facts they can be represented by a combination of a predicate and objects. The predicate refers to a property that shall be represented, e.g. *at* to describe that something is at a certain position. The objects capture between which things this property should hold, e.g. T_L and D. If the predicate is combined with right amount of object something is created, that looks like a previous fact. This will be referred to as atom.

Definition 2: (Predicate)
A predicate p is a tuple (p_{name}, p_{size}). I will also refer to p_{size} as $|p|$. (The arity of p.) A set of predicates is typically denoted by $\mathcal{P}$.

In the following I will refer to the previously mentioned objects. Like facts, objects will just be defined to be any set. You can think of objects as everything we want to put into some relation.

Definition 3: (Atom)
A tuple $(\ p, \vec{o}\ )$ is called an atom with respect to a set of parameters $?X$, predicates $\mathcal{P}$ and objects $\mathcal{O}$ if $p \in \mathcal{P}$, $|\,p\,| = |\,\vec{o}\,|$ and for $i \in \{1, \ldots, |dom_p|\}$ it holds that $o_i \in ?X \cup \mathcal{O}$.

- The atom $(\ p, \vec{o}\ )$ can be also denoted as $p(\ \vec{o}\ )$.
- The set of all such atoms is denoted by $Atoms(?X, \mathcal{P}, \mathcal{O})$.
- A set $Atoms(\emptyset, \mathcal{P}, \mathcal{O})$ is called instantiated.
- If an atom is element of any instantiated set of Atoms, it is called instantiated.

Note that in this definition, atoms does not only capture objects, but also parameters. Parameters will be place holders for objects and will be typically denoted with a question mark in the front (e.g. ?x). Below I will define how a lifted task will be represented with the help of predicates and objects and also how a place holder can be exchanged with a concrete object. This replacement will be called instantiation. This definition of a lifted task should remind you of PDDL syntax.

Definition 4: (Lifted Task)
A Lifted Task is a tuple $(\mathcal{P}, \mathcal{O}, \mathcal{A}_L, \mathcal{I}_L, \mathcal{G}_L)$ where $\mathcal{P}$ is a set of predicates, $\mathcal{O}$ is a set of objects, $\mathcal{A}_L$ is a set of lifted actions, $\mathcal{I}_L \subseteq Atoms(\emptyset, \mathcal{P}, \mathcal{O})$ is the lifted initial state and $\mathcal{G}_L \subseteq Atoms(\emptyset, \mathcal{P}, \mathcal{O})$ is a lifted goal condition.

- An action $a \in \mathcal{A}_L$ is a tuple $(params_a, pre_a, add_a, del_a)$, where $pre_a, add_a, del_a \subseteq Atoms(params, \mathcal{P}, \mathcal{O})$ and $params \cap \mathcal{O} = \emptyset$.
- The set of all possible parameters in an action across a lifted task is denoted by $?X$. I will always assume that in any context it holds that $?X \cap \mathcal{O} = \emptyset$
- Parameters of an action a can be instantiated by replacing parameters with objects, i.e. a set $S \in \{pre_a, add_a, del_a\}$ can be instantiated with the help of a function $f : ?X \cup \mathcal{O} \to \mathcal{O}, f_{|\mathcal{O}} = id$ in the following way:

  $$inst((p, (x_1, ..., x_n)), f) := (p, (f(x_1), ..., f(x_n)))$$

  This way a whole action can be instantiated with such a function f:

  $$inst((params_a, pre_a, add_a, del_a), f) := (f(pre_a), f(add_a), f(del_a))$$

- I'll denote the set of all instantiations by:

  $$Inst(\mathcal{A}_L) := \{inst(a, f) \mid a \in \mathcal{A}_L, f : ?X \cup \mathcal{O} \to \mathcal{O}, f_{|\mathcal{O}} = id\}$$

- One instantiation $inst(a, f)$ can be also denoted by $a(\triangle)$ where $\triangle := (f(?x_1), ..., f(?x_n))$ and $(?x_1, ..., ?x_n) = params_a$.

I will omit the grounded / lifted notation when it is clear from the context.

Example 2: (Grounded Transport Task)
The task from figure 2.1 can be formalized as grounded task in the following way:

- As predicates one needs at least the previously mentioned relations *at* and *in*. Additionally the information that was used in the set-builder notation needs to be captured in some way. This means if to locations are connected and that something is a truck, location or package.

$$\mathcal{P} := \{(at, 2), (in, 2), (connected, 2), (truck, 1), (location, 1), (package, 1)\}$$

- Objects are chosen as the trucks, packages and locations.

$$\mathcal{O} := \{T_L, T_R, P_B, P_R\} \cup Locations$$

- The actions will be defined in a similar fashion as in te grounded task. Note, however, that precondition now captures the information that te set builder condition previously did.

$$\begin{aligned}
&drive := (\\
&\qquad \{?t, ?l_1, ?l_2\}, \\
&\qquad \{at(?t, ?l_1),\ connected(?l_1, ?l_2),\ location(?l_1),\ location(?l_2),\ truck(?t)\}, \\
&\qquad \{at(?t, ?l_2)\}, \\
&\qquad \{at(?t, ?l_1)\}, \\
&)
\end{aligned}$$

$$\begin{aligned}
&load := (\\
&\qquad \{?t, ?p, ?l\}, \\
&\qquad \{at(?t, ?l),\ at(?p, ?l),\ location(?l),\ package(?p),\ truck(?t)\}, \\
&\qquad \{in(?p, ?t)\}, \\
&\qquad \{at(?p, ?l)\}, \\
&)
\end{aligned}$$

$$\begin{aligned}
&unload := (\\
&\qquad \{?t, ?p, ?l\}, \\
&\qquad \{at(?t, ?l),\ in(?p, ?t),\ location(?l),\ package(?p),\ truck(?t)\}, \\
&\qquad \{at(?p, ?l)\}, \\
&\qquad \{in(?p, ?t)\}, \\
&)
\end{aligned}$$

$$\mathcal{A} := \{drive,\ load,\ unload\}$$

- An example instantiation for drive is *drive* is $drive(T_L, S_{T_L}, B) =$

$$
\begin{aligned}
&(\\
&\quad \{at(T_L, S_{T_L}),\ connected(S_{T_L}, B),\ location(S_{T_L}),\ location(B),\ truck(T_L)\},\\
&\quad \{at(T_L, B)\},\\
&\quad \{at(T_L, S_{T_L})\},\\
&)
\end{aligned}
$$

- In the initial state it shall still be captured what the initial positions of trucks and packages are. In addition that it now needs to be added which locations are connected and which objects are a package, a truck or a location.

$$
\begin{aligned}
\mathcal{I}_L \coloneqq\ & \{at(T_L, S_{T_L}),\ at(T_R, S_{T_R}),\ at(P_B, S_{P_B})),\ at(P_R, S_{P_R})\}\\
\cup\ & \{truck(T_L),\ truck(T_R)\}\\
\cup\ & \{location(l) \mid l \in Locations\}\\
\cup\ & \{package(P_R),\ package(P_B)\}\\
\cup\ & \{connected(l_1, l_2) \mid l_1 \text{ and } l_2 \text{ are connected locations}\}
\end{aligned}
$$

- The goal remains that the packages are at their destination.

$$\mathcal{G}_L \coloneqq \{at(P_B, D_{P_B}), at(P_R, D_{P_R})\}$$

Note how similar the notations for the lifted and grounded representation are. I designed the definitions this way on purpose. It should highlight that the difference is rather in the representation of some algorithm, than the understanding of the task (for us humans). Also note that in the grounded representation actions like $drive(T_L, T_L, T_L)$ or $drive(T_L, B, G)$ do not exist, however in the lifted representation they do. One reason for $drive(T_L, T_L, T_L)$ not to exist is that T_L is not a location. The reason for $drive(T_L, B, G)$ is that B and G are not connected. State of the are planner will recognize this (from the information in the initial state) and use this information to ignore/prune such actions. I will also exploit this information in later approaches. (Described in 4.2.)

In the example above I did not describe an example plan. The reason for this is that there is no definition to search in the lifted task. In order to search in the lifted task, it needs to be transferred to a grounded representation. This process is called grounding. Below I will describe a naive way of doing this. In practice grounding is not done in such a naive way, there are many optimizations. E.g. ignoring actions like $drive(T_L, B, G)$ as I described above.

Proposition 1: (Grounding a lifted task)
A Grounded Task $\Pi' = (\mathcal{F}, \mathcal{A}_G, \mathcal{I}_G, \mathcal{G}_G)$ can be obtained from a Lifted Task $\Pi = (\mathcal{P}, \mathcal{O}, \mathcal{A}_L, \mathcal{I}_L, \mathcal{G}_L)$ by setting:

- $\mathcal{F} := Atoms(\emptyset, \mathcal{P}, \mathcal{O})$
- $\mathcal{A}_G := Inst(\mathcal{A}_G)$
- $\mathcal{G}_G := \mathcal{G}_L$
- $\mathcal{I}_G := \mathcal{I}_L$

In order to search with grounded semantics within a lifted representation, recent work introduced lifted successor generation. Intuitively speaking this means that the search only grounds what is necessary for the so far explored states. [1]

2.3 h^{FF}

Before defining h^{FF}, I'll first recap what heuristics and relaxations are.

Definition 5: (Heuristic)
Let S be a set of states. A heuristic (function) $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$ is an estimate for the plan cost of a state. The perfect heuristic (denoted by h^*) returns the minimal plan cost for for a state.

The general idea behind relaxation is to create a task Π' for a task Π that is easier to solve but still has a similar structure. In this case heuristic estimates can be obtained from Π' that show to be helpful for Π.

Definition 6: (Delete Relaxation)
Given a grounded task Π, the delete relaxed task Π^+ is defined by setting $del_a = \emptyset$ for all $a \in \mathcal{A}_G$. I will refer to h^+ for a task Π as the perfect heuristic for its corresponding task Π^+. Note that this is a valid heuristic for Π since the states for Π and Π^+ are the same. h^+ is also called the optimal delete-relaxation heuristic.

A plan for a relaxed task can be computed by simply applying all applicable operands until the goal is reached. However, it should be intuitive that this will not result in good estimates for a real plan. On the other hand it was shown that is NP-hard to compute h^+. So the goal is to find a middle ground that delivers good heuristic estimates, but is not to hard to compute. The planning framework FastForward **??** introduced, among other novelties, a heuristic (h^{FF}) that extracts a more promsing plan.

In the following I describe a way to compute h^{FF}. The computation is divided in two steps. The first one is a forwards exploration in that the information is retrieved, how a fact can be reached. The second one is a backwards exploration, where starting at the goal, a plan is extracted by using the information retrieved before to achieve necessary facts.

1. **Compute a best supporter function:** There are multiple ways to choose a best supporter function. Here I describe a concrete one. In each iteration all actions that are currently applicable are applied. For all newly reached facts, the best supporter is marked as an action that reached the fact. Note that this results in being the action that reached a fact first in this context. This process is repeated until the goal is reached. (See Algorithm 1.)

2. **Extract the delete relaxed plan:** Once the necessary best supporter are computed, the relaxed plan can be extracted. This can done by selecting the best supporter of the goal and recursively the supporter of their preconditions until all preconditions will be fulfilled. (See Algorithm 2.)

Algorithm 1: bs computation

```
1 $C \leftarrow \mathcal{I}_G, bs \leftarrow emptyMap$
2 while $\mathcal{G}_G \nsubseteq C$ do
3     for each $a \in \mathcal{A}$ do
4         if $a$ applicable in $C$ then
5             $C' \leftarrow C[\![a]\!]$
6             for each $f \in C' \setminus C$ do
7                 $bs[f] \leftarrow a$
8             $C \leftarrow C'$
9 return $bs$
```

Algorithm 2: Plan Extraction

```
1 $Open \leftarrow \mathcal{G}_G \setminus \mathcal{I}_G,$
2 $Closed \leftarrow \mathcal{I}_G,$
3 $Rplan \leftarrow \emptyset$
4 while $Open \neq \emptyset$ do
5     select $g \in Open$
6     $Closed \leftarrow Closed \cup \{g\}$
7     $Rplan \leftarrow Rplan\{bs(g)\}$
8     $Open \leftarrow (Open \setminus \{g\}) \cup (pre_{bs(g)} \setminus Close)$
9 return $Rplan$
```

Note that this doesn't extract a plan as defined above. However, this doesn't matter since it yields the amount of actions for the plan what will be the heuristic value. The depicted algorithm is the naive computation. There is an optimization called "counter based", which directly detects if an action a is applicable by tracking the amount if inapplicable preconditions for it. When this amount reaches 0, the action will be marked to be applied in the next iteration. [9]

Chapter 3

Unary Relaxation

In this chapter I will introduce a new relaxation, called Unary Relxation. This relaxation is an addition to the already known delete relaxation. (The task will be even more relaxed.) In this setting, I introduce a plan extraction within a lifted task. Through the delete relaxation it is possible to extract plans in poly-time. And trough the unary relaxation it is possible keep the representation within the (asymptotic) space bounds of the lifted representation.

3.1 Construction

When inspecting the lifted representation of the Transport task, you may notice that there are some predicates for that there are no respective facts in the grounded task. Intuitively one could say that this is not information that the grounded task is missing, but rather information that is already incorporated in the grounded setting. Even if the lifted representation has to store more predicates/information this way, the representation will be smaller in almost all commonly known domains. The representation of facts in the naively grounded representation uses $\theta(\sum_{p\in\mathcal{P}} |\mathcal{O}|^{|p|})$ whereas the lifted representation uses $\theta(|\mathcal{O}| + |\mathcal{P}|)$ space. The space consumption of lifted vs. naively grounded actions shows a similar difference.

Even if this is just the space consumption of the naive grounded representation, which does also not incorporate the previously mentioned information, it should be quite intuitive that on many tasks there is a big difference in the space consumption of the lifted and grounded representation. Note that for this argument to be correct, $\mathcal{I}$ and $\mathcal{G}$ of the lifted task have to stay within the previously mentioned space bounds.

The blowup in space will not be relevant when the arity of all predicates and actions is less or equal to 1. Below I will introduce an action that makes use of this thought by losing the connection between the indexes in predicates. Per convention I will assume that there exists no predicates with arity 0. This is possible since any predicate with arity 0 can be transformed into a predicate of arity 1 by adding atoms with one fixed arbitrary object as instantiation of these atoms.

Definition 7: (Unary Predicate)
A Unary Predicate is a predicate of the form $((p, i), 1)$ where $i \in \mathbb{N}$ and p is element of a set of predicates from another task. A predicate $((p, i), 1)$ is also notated as p_i.

Proposition 2: (Unary Predicate Transformation)
A set of Unary Predicates $UnPr(p) \subseteq (\mathcal{P} \times \mathbb{N}) \times \mathbb{N}$ is obtained from a predicate $p \in \mathcal{P}$ by setting:

$$UnPr(p) \coloneqq \{((p_{name}, i), 1) \mid 1 \leq i \leq |p|\}$$

The set of all unary predicates for a set of predicates $\mathcal{P}$ is denoted by $UnPr(\mathcal{P}) \coloneqq \bigcup_{p \in \mathcal{P}} UnPr(p)$.

Example 3: (Unary Predicates for at)
The set of Unary Predicates $UnPr(at)$ is $\{at_1, at_2\}$ (in the Transport example). So the unary predicates only measure that there exist trucks and packages (with at_1) and that something is at some location (with at_2).

Defining Unary Atoms is straight forward. It's basically just transferring the instantiation/parameter at each index to the according predicate. E.g. $at(T_L, S_{T_L})$ to $\{at_1(T_L), at_2(S_{T_L})\}$ or
$at(?t, B)$ to $\{at_1(?t), at_2(B)\}$.

Definition 8: (Unary Atom)
A Unary Atom is an Atom with a Unary Predicate. A set of Unary Atoms for an Atom($p, \vec{o}$) can be obtained as $UnAt(\ p, \vec{o}\ ) \coloneqq \{(p_i, \vec{o_i}) \mid 1 \leq i \leq |p|\}$. The set of Unary Atoms for a set of Atoms At is defined as $UnAt(At) \coloneqq \bigcup_{at \in At} UnAt(at)$.

The idea behind Unary Relaxed Actions is a little bit more complicated. I will start by defining them.

Definition 9: (Unary Relaxed Action)
A Unary Relaxed Action a is a tuple (P, A) with parameters $P = (p_1, ..., p_n)$ and a set of actions $A = \{a_c, a_{p_1}, ..., a_{p_n}\}$. Action a_c has arity 0. Actions $a_{p_1}, ..., a_{p_n}$ have arity 1. I refer to a_c as the constant part of a and call A the Split Actions of a.

- An instantiation $a_u(o_1, ..., o_n)$ is applicable if $a_c()$ is applicable and $a_{p_1}(o_1), ..., a_{p_n}(o_n)$ are applicable. I.e. $pre_{a_u(o_1,...,o_n)} \coloneqq \bigcup_{1 \leq i \leq n} pre_{a_{p_i}}(o_i) \cup \{pre_{a_c()}\}$.
- The add list add_{a_u} for an instantiation $a_u(o_1, ..., o_n)$ is the join of the according split instantiations, i.e. $add_{a_u(o_1,...,o_n)} \coloneqq \bigcup_{1 \leq i \leq n} add_{a_{p_i}}(o_i) \cup \{add_{a_c()}\}$.
- Unary relaxed actions feature no delete list (or an equivalent to that).
- The arity of a_u is set to the amount of parameters, i.e. $|a_u| \coloneqq |P|$.

The idea behind this definition is that since all predicates are unary, there is at most one parameter in an atom. So the different atoms can be handled through sub-actions, depending on their parameter/instantiation. The constant part of a relaxed action represents all those Unary Atoms that are already instantiated. A Split Action $a_{?x}$ represents all those Unary Atoms that depend on $?x$. This means a transformation can be defined in the following way:

Proposition 3: (Unary Relaxed Action Transformation)
A unary relaxed action $a_u = (params_a, A)$ can be obtained from an action $a = (params_a, pre_a, add_a, del_a)$ in a Lifted Task $\Pi = (\mathcal{P}, \mathcal{O}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ by constructing $A = \{a_c, a_1, ..., a_{|params_a|}\}$ as follows:

- $a_c := (\{p(o) \in UnAt(pre_a) \mid o \in \mathcal{O}\}, \{p(o) \in UnAt(add_a) \mid o \in \mathcal{O}\}, \emptyset)$

- For $1 \leq i \leq |params_a|$:

$$a_{(params_a)_i} := (\{p_i(?x) \in UnAt(pre_a) \mid ?x \in ?X\}, \{p_i(?x) \in UnAt(add_a) \mid ?x \in ?X\}, \emptyset)$$

A transformation for a whole set of actions a is deonted by $ra(A) := \bigcup_{a \in A}\{ra(a)\}$

Example 4: (Unary Relaxed Action for *drive*)
Here I construct the Unary Relaxed action $drive_U := (P, A)$ for the action *drive* from the Transport example. Recall that drive has the parameters $?t, ?l_1$ and l_2, so $P = \{?t, ?l_1, l_2\}$. Thus $A = \{drive_c,\ drive_{?t},\ drive_{?l_1},\ drive_{?l_2}\}$ is constructed as follows:

$drive_c := ($
$\{\},$
$\{\},$
$\{\},$
$\{\}$

$drive_{?t} := ($
$\{?t\},$
$\{truck_1(?t)\},$
$\{at_1(?t)\},$
$\{\}$

$drive_{?l_1} := ($
$\{?l1\},$
$\{location_1(?l_1), connected_1(?l1)\},$
$\{\},$
$\{\}$

$drive_{?l_2} := ($
$\{?l2\},$
$\{location_1(?l_2), connected_2(?l2)\},$
$\{at_2(?l_2)\},$
$\{\}$
$)$

Note that the constant part is empty, since there are no objects in any atoms of *drive*.

By this construction the task is always delete relaxed, since there is no delete list in any action. The reason for this is that when you would define the delete list in a similar fashion as the add list, the relaxed task would become unsolvable at times when the real task is solvable. E.g. by deleting $at_1(T_L)$ when applying $drive_u(T_L, S_{T_L}, B)$ in the example from figure 2.1. So this definition would contradict the intuition behind a relaxation. (Strictly speaking, creating such actions isn't possible, since $eff \cap add \neq \emptyset$.)

If you consider the amount of instantiation that can be used with a unary relaxed action, this is still the same amount as with the original action. However, note that applying some action $a_u(a, b)$ and $a_u(b, a)$ will lead to the same result as applying $a_u(a, a)$ and $a_u(b, b)$. The reason for this phenomenon is that in bot cases the add-lists of a_c, $a_{?a}(a)$, $a_{?a}(b)$, $a_{?b}(a)$ and $a_{?b}(b)$ will be combined. Below I will define a way of applying this at once, denoted as $a_u(\{a, b\}, \{a, b\})$.

Definition 10: (Quick action application)
Given a unary relaxed action a_u and an instantiation $\vec{O}$ with $|\vec{O}| = |a|$ and $\vec{O}_i \subseteq \mathcal{O}$ for $1 \leq i \leq |O|$. $a_u(\vec{O})$ is called applicable when all $a_{(params_a)_i}(o)$ for $1 \leq i \leq |\vec{O}|$, $o \in \vec{O}_i$ and a_c is applicable. The add list of $a_u(\vec{O})$ is the join of all previously listed splits, i.e. $\bigcup_{1 \leq i \leq |\vec{O}|,\, o \in \vec{O}_i} add_{a_{(params_a)_i}}(o) \cup add_{a_c()}$.

Applying the action a_u in some state S means applying $a_u(\vec{O})$ where $\vec{O}$ consists of the maximal sets s.t. $a_u(\vec{O})$ is applicable.

A quick precondition check can be defined in the same way. By applying definition 10 actions can be applied in linear time with respect to the object amount in their instantiation. Applying a_u is (naively) possible in $O(|\mathcal{O}| \cdot |a|)$ by just checking for each index and object if the according split action is applicable. (Whereas there $|\mathcal{O}|^{|a|}$ many different instantiation fore the action.)

Defining the structure of the relaxed task and transformation only consists of combining what was done before:

Definition 11: (Unary Relaxed Task)
A Unary Relaxed Task Π^u is a tuple $(\mathcal{P}_u, \mathcal{O}, \mathcal{A}_u, \mathcal{I}_u, \mathcal{G}_u)$ where:

- $\mathcal{P}_u$ is a set of Unary Predicates.
- $\mathcal{O}$ is a set of objects.
- $\mathcal{A}_u$ is a set of Unary Relaxed Actions.
- $\mathcal{I}_u$ and $\mathcal{G}_u$ are sets of Unary Atoms.

Intuitively the atoms in $\mathcal{I}$ and $\mathcal{G}$ can be interpreted as facts, since they do not contain any parameters. Since being applicable and the add list of a unary relaxed action a_u are defined via their grounded Split Actions, a search can be defined in a similar fashion as in a grounded task.

Definition 12: (Search in a Unary Relaxed Task)
Given a state $s \subseteq Atoms(\emptyset, \mathcal{P}, \mathcal{O})$ within a unary relaxed task:

- An action a can be applied to s when it is called applicable ($pre_a \subseteq s$).
- Applying a to s (denoted by $s[\![a]\!]$) results in a state $s' = s \cup add_a$.
- Applying a sequence of actions $a_1, ..., a_n$ to a state s is defined as $s[\![a_1, ..., a_n]\!] := s[\![a_1]\!][\![a_2, ..., a_n]\!]$.
- A sequence of actions $a_1, ..., a_n$ is called a unary relaxed plan, if $\mathcal{G} \subseteq \mathcal{I}[\![a_1, ..., a_n]\!]$.

Note that the heuristic value of the optimal heuristic, denoted as h^{U*}, in an ultra relaxed task is bounded by h^+ since every delete relaxed plan is also a unary relaxed plan. The task transformation is defined by applying all previously defined transformations.

Proposition 4: (Unary Relaxed Task Tansformation)
A Unary Relaxed Task $\Pi^u = (\mathcal{P}_u, \mathcal{O}, \mathcal{A}_u, \mathcal{I}_u, \mathcal{G}_u)$ can be obtained from a Lifted Task $\Pi = (\mathcal{P}_L, \mathcal{O}, \mathcal{A}_L, \mathcal{I}_L, \mathcal{G}_L)$ by setting:

- $\mathcal{P}_u := UnPr(\mathcal{P}_L)$
- $\{a_u \mid a \in \mathcal{A}\}$
- $\mathcal{I}_u := UnAt(\mathcal{I}_L)$
- $\mathcal{G}_u := UnAt(\mathcal{G}_L)$

Example 5: (Unary Relaxed Goal and Initial State)
Previous examples already showed how to obtain a unary predicate and action in the running example. Here I want to extract the unary goal and inital state, since I will refer to them later on.

- $$\begin{aligned} \mathcal{I} := & \{at_1(P_B,),\ at_2(S_{P_B}),\ at_1(P_R),\ at_2(S_{P_R})\, at_1(T_L,),\ at_2(S_{T_L}),\ at_1(T_R),\ at_2(S_{T_R})\} \\ \cup\ & \{connected_1(l) \mid l \in Locations\} \\ \cup\ & \{connected_2(l) \mid l \in Locations\} \\ \cup\ & \{truck_1(T_L),\ truck_1(T_R)\} \\ \cup\ & \{location_1(l) \mid l \in Locations\} \\ \cup\ & \{package_1(P_R),\ package_1(P_B)\} \end{aligned}$$
- $\mathcal{G} := \{at_1(P_B),\ at_2(D_{P_B}),\ at_1(P_R),\ at_2(D_{P_R})\}$

3.2 Plan Extraction

The plan extraction is done in a similar way as in h^{FF}. It starts with a forward iteration where the best supporter function is computed and extracts a plan with that. The best supporter function here maps unary atoms to split actions.

For the best supporter computation the main difference, apart from using atoms instead of facts, is that now the first possible instantiation of each parameter in an action is additionally tracked. This will be denoted by the *first* function. Why this is necessary becomes clear when looking at the plan extraction. To highlight differences I marked the modified parts (of the original h^{FF} algorithm) in blue.

Algorithm 3: bs computation

```
$C \leftarrow \mathcal{I}, bs \leftarrow emptyMap$
$first \leftarrow map(a_i \mapsto none \mid a \in \mathcal{A} \land 1 \leq i \leq |a|)$
while $\mathcal{G} \not\subseteq C$ do
    for each $a \in \mathcal{A}$ do
        for each $a_x \in a_{splits}, o \in \mathcal{O}$ do
            if $a_x(o)$ applicable in $C \land first[a_x] = none$ then
                $first[a_x] \leftarrow o$
        if $a$ applicable in $C$ then
            for each $a_x \in a_{splits}$ do
                $C' \leftarrow C[\![a_x]\!]$
                for each $p_i(o) \in C' \setminus C$ do
                    $bs[p_i(o)] \leftarrow a_x$
            $C \leftarrow C'$
return $bs$
```

I already mentioned that for the plan extraction we need the *first* function. Consider the following example to get an idea why this is necessary:

Example 6: (The unary relaxed plan extraction problem)
When extracting a plan for the running example, $at_2(D_{P_B})$ and $at_2(D_{P_R})$ still need to be achieved. (The rest of the goal is true in the initial state.) Assume $bs(at_2(D_{P_B})) = drive_{?l2}(D_{P_B})$. So the plan will contain $drive_u(\_, \_, D_{P_B})$, but it is not clear, what the blanks should be.

As solution for this problem I choose the first possible instantiation of the corresponding parameter in an action. The intuition behind this is that this should not introduce too much overhead, since this parameter will be achieved "quickly". Following this idea, the rest of the plan extraction is straight forward. The open list will consist of unary atoms. Step by step the elements that are (still) in the open list are select. For each element the best supporter is determined. For the best supporter a action instantiation can be selected with the help of the first function. The precondition of this action instantiation is added to the open list and the effect is marked as achieved.

This time I did not mark common/different parts, since the syntax of both plan extraction algorithms differs too much. However the general structure of both algorithms is still the same.

Algorithm 4: Plan Extraction

```
1  $Open \leftarrow \mathcal{G} \setminus \mathcal{I}$,
2  $Closed \leftarrow \mathcal{I}$,
3  $Rplan \leftarrow \emptyset$
4  while $Open \neq \emptyset$ do
5      select $p_j(o) \in Open$
6      $Closed \leftarrow Closed \cup \{p_j(o)\}$
7      $a_x \leftarrow bs[p_j(o)]$
8      $inst \leftarrow tuple\ of\ size\ |a|$
9      for each $i \in \{1, ..., |a|\} \cup \{c\}$ do
10         $o' \leftarrow o$ if $x = (params_a)_i$ else $first(a_i)$
11         $a_x \leftarrow a_{(params_a)_i}$
12         $Open \leftarrow Open \cup pre_{a_x(o')}$
13         $Closed \leftarrow Closed \cup eff_{a_x(o')}$
14         if $i \neq c$ then
15             $inst[i] = o'$
16     $Open \leftarrow (Open \setminus Closed)$
17     $Rplan \leftarrow Rplan \cup \{a(inst)\}$
18 return $Rplan$
```

I will refer to the heuristic that is computed this way as $h^{U_{ff}}$.

Example 7: (Unary Relaxed Plan)
A unary relaxed plan for the running example is $drive_u(T_L, S_{T_R}, D_{P_B})$, $drive_u(T_L, S_{T_R}, D_{P_R})$. It is possible to apply these actions, since *connected*$_1$ and *connected*$_2$ is true for any location and $at_1(S_{T_R})$ is initially true. Since $drive_u(T_L, S_{T_R}, D_{P_B})$ achieves $at_2(D_{P_B})$ and $drive_u(T_L, S_{T_R}, D_{P_R})$ achieves $at_2(D_{P_R})$ the goal is fulfilled.

Chapter 4

Refining the approach

As you may have already realized, the unary relaxation loses a lot information. This raises the question how good the heuristic estimates are, which I will partially answer by doing a first evaluation. This evaluation will give a rough idea of potential pitfalls. I will address some pitfalls in more detail and introduce potential solutions for them.

4.1 An initial evaluation

The technical details and improvements of the concrete implementation are described in 5.2. Since the ambition is to improve the performance of the planner, I compare $h^{U_{ff}}$ to Goal Counting (GC) and breadth first search (BFS), which are both search options that are currently available. The tasks used are from past optimal IPCs.

Definition 13: (Goal Counting)
Goal Counting is the heuristic h^{GC} that counts all unachieved goals.

$$h^{GC} : S \to \mathbb{N}, s \mapsto |\mathcal{G} \setminus s|$$

When inspecting the coverage table in figure 4.1 you can see that there are three domains; namely Agricola, Childsnack and PNetAlignment; which are were completely unsolved by GC and BFS, but $h^{U_{ff}}$ was at least able to solve some instances. In PNetAlignment $h^{U_{ff}}$ even solves all instances. So in terms of being able to solve new domains, $h^{U_{ff}}$ already achieved something new. When comparing the overall performance within coverage, however, GC solves 239 instances more than $h^{U_{ff}}$, which is the result of many domains where GC solves more instances.

Since $h^{U_{ff}}$ has a huge overhead in computation time compared to GC, it is interesting to know, how informed the search is. To investigate this, I plotted the amount of expansion of GC vs. $h^{U_{ff}}$ in a scatter plot. (Figure 4.2a) As you can see, this gives no clear result. To make a little more sense of this expansions within different domains are illustrated in figure 3 of the appendix. This shows how domain dependent the different performance of the heuristics is. The reason for this is probably the different natures of the heuristic. While GC is only concerned about the goal, $h^{U_{ff}}$ is not goal aware (h(s) = 0 iff s is a goal).

To get an idea if the heuristic is informative in general, I plotted the amount of expansions in all commonly solved instances of $h^{U_{ff}}$vs. BFS. I colorized those domains, where the domain wise difference in expansions was the highest. Here you can observe that for

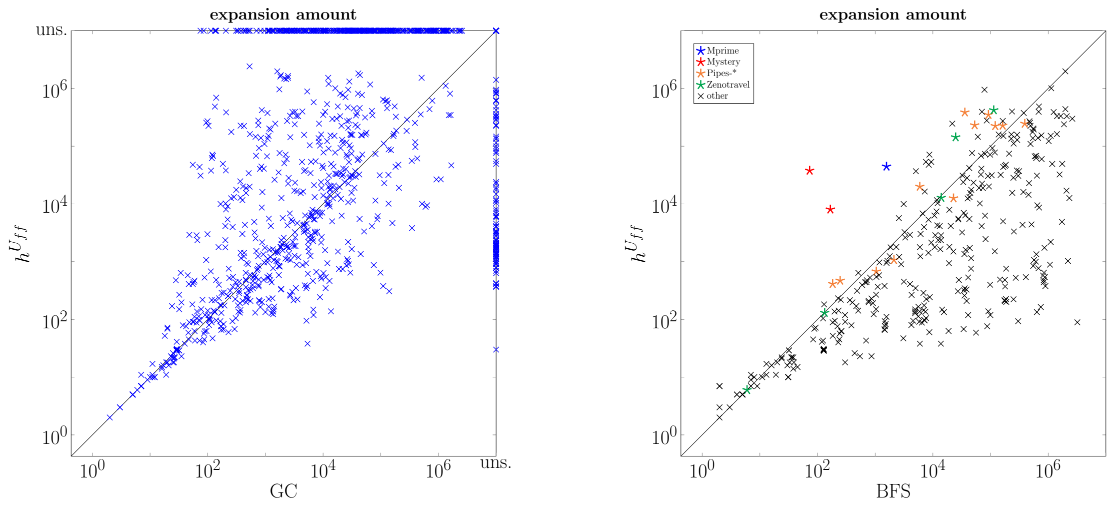
Domain	BFS	GC	h^{Uff}
Agricola (20)	0	0	4
Airport (50)	16	**22**	19
Barman (34)	0	**18**	4
Blocks (35)	15	**35**	17
Childsnack (20)	0	0	**3**
DataNetwork (20)	9	15	14
Depot (22)	2	**11**	3
DriverLog (20)	5	**15**	5
Elevators (30)	7	**30**	12
Floortile (40)	0	**2**	0
Freecell (80)	13	30	42
GED (20)	15	**20**	17
Grid (5)	1	**2**	0
Gripper (20)	6	**20**	6
Hiking (20)	8	8	**9**
Logistics (63)	11	**30**	11
Miconic (150)	40	**150**	134
Movie (30)	**30**	**30**	**30**
Mprime (35)	14	**18**	1
Mystery (19)	11	**13**	2
Nomystery (20)	6	**9**	6
Openstacks (80)	12	37	**77**
OrgSynth (20)	10	**12**	**12**
OrgSynth-split (20)	6	6	**8**
Parcprinter (30)	7	**14**	8
Parking (40)	0	**2**	0
Pathways (30)	3	5	5
Pegsol (36)	31	**36**	35
Pipes-notank (50)	11	**32**	6
Pipes-tank (50)	6	**18**	7
PNetAlignment (20)	0	0	**20**
PSR (50)	39	**46**	**46**
Rovers (40)	4	15	**17**
Satellite (36)	2	**6**	4
Scanalyzer (30)	6	**30**	13
Snake (20)	1	5	**6**
Sokoban (30)	9	14	**15**
Termes (20)	3	**15**	2
Tetris (17)	5	**17**	15
Tidybot (40)	1	**26**	1
TPP (30)	5	**10**	6
Transport (60)	13	**58**	21
Trucks (30)	1	2	**9**
VisitAll (40)	7	**40**	**40**
Woodworking (30)	5	**27**	7
Zenotravel (20)	5	**13**	6
Sum (1622)	401	**964**	725

Figure 4.1: A comparison of breadth first search (BFS), Goal Counting (GC) and the newly introduced heuristic (h^{Uff}) in all solvable instances from the optimal tracks of the International Planning Competitions up to 2018, that do not have conditional effects or axioms.

some domains the heuristic is "misleading", even if it seems to be informative for most domains.. In the following sections I will investigate some of these problems.

4.2 Static Unrelaxation

I've already mentioned a lot of times that within the Transport task the predicates *connected*, *truck*, *package* and *loacation* play a different role than *at* and *in*. Here we will categorize them as static. Intuitively speaking, static information is the information that



(a) GC vs h^{Uff} **(b)** BFS vs. h^{Uff}

Figure 4.2: Comparison of expansions within the different searches.

does not change while exploring the task. Static predicates are widely recognized to be important for effective grounding. [6] While grounding a lot of actions can be rendered to be inapplicable just form the static information. Example 7 shows that within the unary relaxation, any location can be reached in the running example. Figure 4.3 illustrates really nicely, why this is a problem.

Definition 14: (Static Predicates)
A predicate $p \in \mathcal{P}$ is called static, if there exits no action $a \in \mathcal{A}$, $\vec{o} \in ?X \cup \mathcal{O}^{|p|}$ s.t. $p(\ \vec{o}\ ) \in eff_a$.

- The set of all static predicates is called stPr.
- An atom $p(\ \vec{o}\ )$ is called static if p is static.
- The set of all static atoms within $\mathcal{I}$ is called stAt.

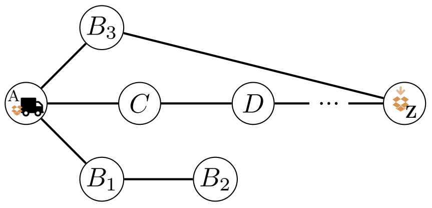


Figure 4.3: An illustration of a NoMistery task with one truck and one package. The truck already picked up the package and is currently in A. The package needs to be delivered to Z.

Example 8: (No distance measurement)
The search of the actual task, depicted in figure 4.3, has 4 successor states for which the heuristic will be queried for:

- Drive the truck to B_3.
- Drive the truck to B_1.
- Drive the truck to C.
- Drop the package at A.

In the unary relaxed task, $connected_1(l)$ and $connected_2(l)$ is true for every $l \in Locations$. This means any location can be reached within one $drive_u$ action. So for any state the heuristic value will be one, since the goal can still be reached with $drive_u(T, A, Z)$. The heuristic does not capture the following properties of the task:

1. B_1 leads away from Z.
2. The road over B_3 is shorter than any other road.
3. The package should be in the truck to travel.

Here I want to introduce an extension that addresses the listed problem 1 and 2 in example 8. I want to do this by introducing additional preconditions to unary relaxed actions. I will do an example computation of this additional preconditions first.

Example 9: (Computing Additional Preconditions)
Consider the *drive* action from the running example. And the task depicted in figure 4.3. Assume that $at(T, C)$ is achieved. Then $?t$ has to be instantiated with T and $?x$ with C. Since the precondition contains $connected(?x, ?y)$, $?x$ needs to be instantiated with some X in a way s.t. $connected(X, C)$ is fulfilled. From stAt it can be concluded that there are two options for $?x$: A and D. This means in order to achieve $at(T, C)$, $at(?t, A)$ or $at(?t, D)$ has to be true. This requirement is useless in the actual task, since it is always true when the precondition is fulfilled. However transferring this to the lifted unrelaxation you can now require $pre_{drive_{?x}(A)}$ or $pre_{drive_{?x}(D)}$ to be $\subseteq s$ in order for $drive_{?y}(C)$ to be applicable in some state s. $pre_{drive_{?x}(A)}$ and $pre_{drive_{?x}(D)}$ can be used here, since they denote exactly those preconditions where $?x$ was substituted by D.

In algorithm 5 a way of computing this additional preconditions in a generic way is introduced. The idea behind this algorithm is the same as the idea described in example 9. Note that this additional preconditions are actually conditions (logical) terms, opposed to just sets as normal preconditions. Strictly speaking $\wedge$ and $\vee$ are not defined for atoms. The definition however, would be straight forward by interpreting an atom a as true iff it is element of the current state. To reduce the runtime overhead of this extraction, the relations are just quadratic, i.e. this check is only done for pairs of positions/objects.

Algorithm 5: Quadratic Extraction of Additional Pres of $a_x(o)$

```
1  $add\_pre \leftarrow true$
2  for each static Atom $p(\vec{o}) \in pre_{a_x}$ do
3      for each $i$ with $\vec{o}_i = x$ do
4          for each $j \neq i \wedge 1 \leq j \leq |p|$ do
5              if $\vec{o}_j \in \mathcal{O}$ then
6                  if $\neg\exists p(\vec{o'}) \in stAt : \vec{o'}_i = o \wedge \vec{o'}_j = \vec{o}_j$ then
7                      return $false$
8              else
9                  $add\_pre \leftarrow add\_pre \wedge (\bigvee_{o' \in \mathcal{O}} pre_{a_{\vec{o}_j}(o')})$
10 return $add\_pre$
```

The check in line 3 can be changed to $\vec{o}_i \in \mathcal{O}$ to run use the algorithm with a_c. To incorporate the additional precondition within a relaxed task, Definition 9 would need to be changed to: An instantiation $a_u(o_1, ..., o_n)$ of a Unary Relaxed Action a_u is applicable in a state s if the following two statements are true:

- The additional precondition of a_c evaluates to true in s and $pre_{a_c()} \subseteq s$.
- For each $1 \leq i \leq |a_u|$: $pre_{(params_a)_i}(o_i) \subseteq s$ and the additional precondition of $pre_{(params_a)_i}(o_i)$ evaluates to true.

This basically means that both the additional precondition and normal precondition have to be true in the current state.

4.3 Predicate Splitting

In previous examples the extracted plans in the unary relaxation only consisted of $drive_u$ actions. This is the case because in the unary relaxation it is only captured that something is at a certain position, not what. Here I want to introduce an extension s.t. at least a package has to be at this position instead of a truck.

Example 10: (Perfect for Predicate Splitting)
In the task, depicted in figure 4.4, the goal is already achieved within the unary relaxation. Now assume that within Logistics tasks the location of packages would be determined via a predicate *at_package* and the location of trucks via a predicate *at_truck*. Then also within the unary relaxation the task is not solved yet since only $at\_truck_2(D_{P_R})$ is fulfilled, but not $at\_package_2(D_{P_R})$. In this particular case it holds that $h^+ = h^{U*}$.

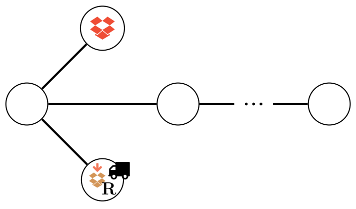


Figure 4.4: An illustration of a NoMistery task with one truck and one package. The truck did not deliver the package yet and is currently in the destination of the package.

Intuitively splitting up the *at* predicate as proposed in example 10 is possible, since trucks and packages are completely different things and so are handled separately. "Being handled separately" is an important point here. For all atoms $at(?x, ?l)$ in $pre_a \cup eff_a$ of some action a, $?x$ is constrained to either $truck(?x)$ or $package(?x)$ in pre_a. In the task there is no object that fulfills *truck* and *package* at the same time. So the *at* atoms will always have different instantiations and thus can be consequently renamed without changing the semantics of the task. In the next extension you can see that similar things are also possible when the atoms share instantiations. However, this would introduce additional actions. Here only one predicate is added to the task, which is a negligible addition in terms of space. In the following I introduce an algorithm that detects such predicates and "renames" them.

One equivalence class, computed by algorithm 6, is the join of all possible instantiations that share at least one atom. Note that a equivalence relation has to be transitive So if $p(\vec{o}_1)$ and $p(\vec{o}_2)$ share one instantiation, $p(\vec{o}_2)$ and $p(\vec{o}_3)$ share one instantiation, but $p(\vec{o}_3)$ and $p(\vec{o}_1)$ don't, their instantiations are still combined to on equivalence class. Something is considered to be a possible instanatiation when the instantiation does not contradict any static atom in the precondition with arity 1. The equivalence classes for *at* in the running example are $\{(t, l) \mid t \in \{T_L, T_R\}, l \in locations\}$ and $\{(p, l) \mid t \in \{P_B, P_R\}, l \in locations\}$. The renaming is done with help of these equivalence classes.

Definition 15: (Instantiation Equivalence Classes)
Algorithm 6 describes a way to extract the instantiation equivalence class of objects for a predicate $p \in \mathcal{P}$. This will be denoted as $\mathcal{P}/[\![\cdot]\!]_p$. Where $[\![\cdot]\!]_p$ is the implicit equivalence

Algorithm 6: Extract object equivalence classes for p

```
1  $EquvClasses \leftarrow \emptyset$
2  for each $a \in \mathcal{A},\ p(\vec{o}) \in pre_a \cup eff_a$ do
3      $new\_class[|p|]$
4      for each $1 \leq i \leq |p|$ do
5          $sp \leftarrow \{p \in stPr \mid |p| = 1 \wedge p(\vec{o_i}) \in pre_a\}$
6          $new\_class[i] \leftarrow \{o \in \mathcal{O} \mid \forall p \in sp : p(o) \in stAt\}$
7      for each $cl \in EquvClasses$ :
8        $\forall 1 \leq i \leq |p| : new\_class[i] \cap cl[i] \neq \emptyset$ do
9          $EquvClasses \leftarrow EquvClasses \setminus cl$
10         for each $1 \leq i \leq |p|$ do
11             $new\_class[i] \leftarrow new\_class[i] \cup cl[i]$
12     $EquvClasses \leftarrow EquvClasses \cup new\_class$
13 return $EquvClasses$
```

relation. The join of all so computed equivalence classes $\bigcup_{p\in\mathcal{P}} \mathcal{P}/[\![\cdot]\!]_p$ is denoted by $\mathcal{P}/[\![\cdot]\!]$.

To "rename" predicates, the predicates become a tuple of the old predicate and one equivalence class. In the following I define a function that returns the new predicate for an atom.

Definition 16: (Split Function)
The Split Function spf is defined in the following way:

$$spf : Atoms(\emptyset, \mathcal{P}, \mathcal{O}) \rightarrow \mathcal{P} \times \mathcal{P}/[\![\cdot]\!],$$

$$p(o) \mapsto (p, [\![o]\!]_p)$$

With this, transforming atoms without parameters will be straight forward. However transforming the atoms within action needs some extra care, since this has to be done with reference to the precondition of the according action. In the following I define an algorithm that creates a function that maps parameters to a substantiation that matches the constraints introduced by the static atoms with arity 1 in the precondition of some action a.

Algorithm 7: Action substitution for $a \in \mathcal{A}$

```
1  $obj\_map \leftarrow \{o \mapsto o \mid o \in \mathcal{O}\}$
2  $constraints \leftarrow \{p \mapsto \ \emptyset \mid p \in ?X\}$
3  for each static atom $p(?x) \in pre_a$ with $|p| = 1 \ \wedge \ ?x \in ?X$ do
4      $constraints[?x] \leftarrow constraints[?x] \cup \{p\}$
5  return $\{o \mapsto o \mid o \in \mathcal{O}\}$
6    $\cup \{?x \mapsto \ o \mid ?x \in ?X, o \in \mathcal{O}, \forall p \in constraints[?x] : p(o) \in stAt\}$
```

Definition 17: (Parameter Substitution)

A function $psubst_a : ?X \cup \mathcal{O} \to \mathcal{O}$ for some $a \in \mathcal{A}$ is defined as the result of algorithm 7.

This valid instantiation mapping can be used to determine the predicate. The following function does that and directly places it into an atom with the same instanatiation as the input.

Definition 18: (Condition Transformation)
The Condition Transformation function for an action $a \in \mathcal{A}$ is defined in the following way:

$$ctr_a : Atoms(?X, \mathcal{P}, \mathcal{O}) \to Atoms(?X, \mathcal{P}, \mathcal{O}),$$
$$p(o_1, ..., o_n) \mapsto (spf(p(psubst_a(o_1), ..., psubst_a(o_n))), (o_1, ..., o_n))$$

The task transformation then combines all previous definitions:

Proposition 5: (Predicate Splitting)
A Partial Instantiation Lifted Task $\Pi^{SP} = (\mathcal{P}_{SP}, \mathcal{O}, \mathcal{A}_{SP}, \mathcal{I}_{SP}, \mathcal{G}_{SP})$ can be obtained from a Lifted Task $\Pi = (\mathcal{P}_L, \mathcal{O}, \mathcal{A}_L, \mathcal{I}_L, \mathcal{G}_L)$ by setting:

- $\mathcal{P}_{SP} \coloneqq \mathcal{P}_L \times \mathcal{P}_L / [\![\cdot]\!]$
- $\mathcal{A}_{SP} \coloneqq \{(params_a, ctr(pre_a), ctr(add_a),\ ctr(del_a)) \mid a \in \mathcal{A}_L\}$
- $\mathcal{I}_{SP} \coloneqq \bigcup_{p(\vec{o}) \in \mathcal{I}_L} \{(spf(p(\vec{o})), \vec{o})\}$
- $\mathcal{G}_{SP} \coloneqq \bigcup_{p(\vec{o}) \in \mathcal{G}_L} \{(spf(p(\vec{o})), \vec{o})\}$

4.4 Partial Instantiation

With predicate splitting it is now possible to detect that a package is in a certain location within a Transport task. Still, it is not possible to differentiate which package is at a certain location. Here will again introduce an extension that addresses this problem via a task transformation.

Example 11: (Perfect Package Instantiation)
In the task, depicted in figure 4.5, the goal is already achieved within the unary relaxation. Now assume that within this task the location of package P_B and P_r is determined via two different predicates at_P_R and at_P_B. Then also within the unary relaxation the task is not solved yet, since only $at\_P_R(D_{P_B})$ and $at\_P_B(D_{P_R})$ is fulfilled, but not $at\_P_B(D_{P_B})$ and $at\_P_R(D_{P_R})$. In this particular case it holds that $h^+ = h^{U*}$.

The problem with such a transformation is that it does not only introduce additional predicates, but also additional actions.

Example 12: (Action blow up with partial instantiation)
Revisit the action *load*. The precondition contains the predicates $at(?t,\ ?l)$ and $at(?p,\ ?l)$. Ignoring any further analyses, $?t$ and $?p$ can both be instantiated with P_R. So when introducing at_P_R to track the position of P_R you need to consider 4 cases:

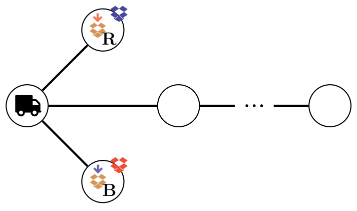


Figure 4.5: An illustration of a NoMistery task with one truck and two packages. The truck did not deliver any package. package R(ed) is currently in the destination of package B(lue). And package B(lue) is currently in the destination of package R(ed).

- Neither $?t$ and $?p$ are instantiated with P_R.
- Only $?t$ is instantiated with P_R.
- Only $?p$ is instantiated with P_R.
- Both $?t$ and $?p$ are instantiated with P_R.

Which are all addressed by different combinations of the predicates at and at_P_R, this is why 4 actions are needed instead of one. In this particular example it could be detected that $?t$ can not be instantiated with at_P_R. This would be implicitly the case when applying Predicate Splitting before, since $?t$ then wouldn't be considered as potential parameter. In general however, it is possible that blow up of newly introduced actions is even bigger.

Still it may be worth to accept this blow up in space in really rare cases when the gain of information is big enough. Here I want to introduce a task transformation that basically pre-instantiates one particular position of one one predicate with a particular object. In order for the relaxation to "recognize" this, it will be done by renaming the predicate, similar to the way in the previous example. The following transformation will be fixed for $p \in \mathcal{P}$, $o \in \mathcal{O}$ and $i \in \mathbb{N}$. As new predicate name I will use (p, o, i) to denote that the predicate p was instantiated at position i with object o.

Definition 19: (Partially Instantiated Predicate)
As $\mathcal{P}_{inst}$ I denote $\mathcal{P}$ extended by the newly introduced predicate:

$$\mathcal{P}_{inst} := \mathcal{P} \cup \{p_{inst}\}$$

Where inst p_{inst} is the newly introduced predicate

$$p_{inst} := ((p, o, i), |p| - 1)$$

For convenience, I'll define a function that removes an element from a vector, before defining the actual transformation function.

Definition 20: (Truncate)
The truncate function for a vector $\vec{o} \in \mathcal{O}^n$ with $n \in \mathbb{N}^*$ is defined as:

$$tr_n : \mathcal{O}^n \to \mathcal{O}^{n-1},$$

$$((o_1, ..., o_n), i) \mapsto \begin{cases} (o_2, ..., o_n) \text{ , if } i = 1 \\ (o_1, ..., o_{n-1}) \text{ , if } i = n \\ (o_1, ..., o_{i-1}, o_{i+1}, ..., o_n) \text{ , otherwise} \end{cases}$$

The usage of $tr(\vec{o})$ implicitly refers to $tr_{|\vec{o}|}(\vec{o})$.

To handle the permutations I will use decisions function $f : ?X \to \{true, false\}$ that decide if a parameter should be instantiated or not. Below I will define a Transformation Function that transforms atoms with predicate p and ignores every other predicate.

Definition 21: (Transformation Function)
The Transformation Function tf_f, a is defined with respect to a function $f : ?X \to \{true, false\}$ and an action $a \in \mathcal{A}$ in the following way:

$$tf_f : Atoms(?X, \mathcal{P}, \mathcal{O}) \to Atoms(?X, \mathcal{P}_{inst}, \mathcal{O})$$

$$p'(\ \vec{o}\ ) \mapsto subst(x),\ x = \begin{cases} p_{inst}(tr(\vec{o}, i)), \text{ if } p' = p \wedge \vec{o}_i = o \\ p_{inst}(tr(\vec{o}, i)), \text{ if } p' = p \wedge \vec{o}_i \in ?X \wedge f(\vec{o}_i) \\ p'(\ \vec{o}\ ), \text{ otherwise} \end{cases}$$

Where *subst* is the function that replaces all parameters $?x$ in an atom with when for that it holds that $f(?x) \wedge \exists p'(\vec{o}) \in pre_a : \vec{o}_i = ?x \wedge p' = p$. Meaning tat $?x$ occurred once at position i of an atom with predicate p. I'll denote $(f, a) = \varepsilon$, in cases where it does not matter what f is chosen. This is the case if $?X = \emptyset$.

Proposition 6: (Partial Instantiation)
A Partial Instantiation Lifted Task $\Pi^{PG} = (\mathcal{P}_{PG}, \mathcal{O}, \mathcal{A}_{PG}, \mathcal{I}_{PG}, \mathcal{G}_{PG})$ can be obtained from a Lifted Task $\Pi = (\mathcal{P}_L, \mathcal{O}, \mathcal{A}_L, \mathcal{I}_L, \mathcal{G}_L)$ by setting:

- $\mathcal{A}_{PG} \coloneqq \bigcup_{a \in \mathcal{A}_L} \{(params_a,$
  $tf_{f,a}(pre_a) \cup \{allow(?x)\ |?x \in ?X, \neg f(?x) \wedge \exists p'(\vec{o}) \in pre_a : \vec{o}_i = ?x \wedge p' = p\},$
  $tf_{f,a}(add_a),\ tf_{f,a}(del_a) \mid f : ?X \to \{true, false\}\}$
- $\mathcal{P}_{PG} \coloneqq (\mathcal{P}_L)_{inst} \cup \{allow\}$
- $\mathcal{I}_{PG} \coloneqq tf_\varepsilon(\mathcal{I}_L)\ \cup\ \{allow(o') \mid o' \in \mathcal{O} \setminus \{o\}\}$
- $\mathcal{G}_{PG} \coloneqq tf_\varepsilon(\mathcal{G}_L)$

The construction above seems really complicated, however the idea is simple. Say for each action a the parameters $?P$ are determinted to be potentially instantiated. Now an action is created for each $x \in 2^{?P}$. (All possible combinations of instantiating/not instantiating a parameter.) When a parameter $?x$ is not instantiated, $allow(?x)$ is added to the precondition of the according action. This atom is used to forbid the instantiation of $?x$ with o in this action. ($allow$ is true for every object except o.)

Chapter 5

Results

Here I will do an evaluation of all previously described extensions to the original approach. I start by listing implementation specific details and then continue to analyze the benchmarks I've run. To conclude, what I think was not captured by the benchmarks, I introduce a new hard to ground domain.

5.1 Implemtation specific optimizations

Negated preconditions:
The lifted framework is still in an early stage and therefore does not provide many features. E.g. it does not support conditional effects or negated preconditions, i.e. single atoms that should not be true in the current state in order for the action to be applicable. To support the later, I introduce a naive transformation. Note that negations are also not introduced within the formalism of this paper. However, they appear in many benchmarks and so are interesting to support. The transformation substitutes every negated atom $p(\vec{o})$ that appears in some precondition with a non-negated atom $neg\_p(\vec{o})$ (in this precondition). Additionally some $neg\_p(\vec{o})$ is added to the add list of an action when $p(\vec{o})$ occurs in the delete list of this action. Some $neg\_p(\vec{o})$ is added to the delete list of an action when $p(\vec{o})$ occurs in the add list of this action. And for all $\vec{o}$, $neg\_p(\vec{o})$ is added to $\mathcal{I}$if $p(\vec{o})$ is not part of the initial state. Especially adding that many atoms to $\mathcal{I}$has the potential of introducing a big overhead. However in all benchmarks the arity of negated atoms in a precondition is ≤ 3. In no benchmark the transformation took longer than 1 minute which is a neglectable overhead within the 30 minute time-frame for the benchmarks.

Types:
Types are completely omitted in the formal definition of this paper. However, they are defined in all commonly used PDDL domains. You can think of the types as the static predicates of arity that were exploited before, the reason behind this is that they originate from the respective types. In many definitions and propositions of the paper I consider any object as instantiation for any parameter. In reality (i.e. the actual implementation) this is not the case. There I will always only consider the objects for the respective type of that parameter.

I also use types for the implementation of algorithm 6. The common object object can be done by checking if there is an object that has the type of both parameters (through inheritance), By this parameters can be used for the check instead of objects and an inheritance check instead of a subset check. This is presumably a lot faster. Note that this does potentially not capture all static predicates of arity 1. Namely, if such a predicate does not originate from a type. However, in reality this is rarely the case.

Counter based approach:
Similar to the counter based approach in h^{ff}, I implemented a counter based approach in the forward iteration within the unary unrelaxation.

For the original configuration this is straight forward by having one counter for the precondition of each split action and one counter for amount of split actions for that there exists a valid instantiation already.

Within the static unrelaxation an additional counters are needed to track if an additional preconditions are true. Meaning that these counters track the amount of clauses of the outer most conjunction that do not yet evaluate to true. Since these clauses are disjunctions, this can be done by tracking that one precondition within in the disjunction became true. Note that this the case when in the original relaxation the action becomes applicable, so this mechanism can be used. Additionally, a counter, like the counter for normal preconditions is needed, to track the amount of split actions for that there exists an instantiation s.t. the additional precondition is true.

5.2 Evaluation

I implemented my approaches on top of the Power Lifted Planner [3]. All task transformations were added within the translation unit, the heuristic were plugged into the already existing greedy best first search. In all searches Yannakakis' Algorithm [14] was used as as successor generation. The experiments were run using the Lab framework [13] on a cluster of machines with Intel Xeon E5-2650 CPUs with a clock speed of 2.30GHz. Each instance of an experiment was run with time and memory limits of 30 minutes and 4 GB respectively. The following benchmark suites were used:

1. All solvable instances from the optimal tracks of the IPCs up to 2018 that do not have conditional effects, either typed predicates or axioms. Resulting in a total of 1622 unique instances from 46 domains. Note that most of this domains can be grounded without a problem by state-of-the art planners. However, these benchmarks offer a wide variety of domains, which is not the case for hard to ground benchmarks, so far. I decided for the optimal benchmark set, even if the planner is a satisfying planner, since the performance is expected to be rather bad, compared to state-of-the-art planner, these benchmarks were designed for. The reason for this is that state-of-the art planner have been optimized in many ways which was not the case for the lifted planning framework so far. I believe that these benchmark set is the best currently available option to get an idea about the quality of the heuristic.

2. The set of hard to ground benchmarks from "Lifted Successor Generation using Query Optimization Techniques" [1], to compare the already currently available options of the framework. This totals 418 task on 6 domains.

For comparison with already existing grounded techniques, I used the h^{max}, h^{FF} and GC implementations of Fast Downward [5] (also combined with GBFS). The experiments were also run on the same cluster, using the Lab framework and with same time and memory

configuration	description
BFS	Breadth first search
GC	Goal counting (in the Power Lifted Planner)
$h^{U_{ff}}$	h^{FF}-like extraction within the unary relaxed task
$h^{U_{max}}$	h^{max}-like extraction within the unary relaxed task
$h^{U^2_{ff}}$	h^{FF}-like extraction within the unary relaxed task (with static unrelaxation)
n $h^{U^2_{max}}$	h^{max}-like extraction within the unary relaxed task (with static unrelaxation)
$spl(\cdot)$	denotes that predicate splitting is applied before
$pinst(\cdot)$	denotes that partial instantiation is applied before
$greedy(\cdot)$	denotes that during the plan extraction the greedy strategy is used
h^{FF}	h^{FF} in Fast Downward
h^{maxx}	h^{max} in Fast Downward
GC_{FD}	Goal Counting in Fast Downward

Figure 5.1: Abbreviations for all benchmark configurations.

limits. I will start by assessing the different approaches within the IPC benchmarks. This was already partially done in 4.1. The coverage scores are obtained from the coverage table in the appendix. I will refer to the configurations by the abbreviations introduced in figure 5.1. In this list there are configurations that I did not define before. I will do this in the following:

Definition 22: (h^{max})
h^{max} is a heuristic. for a given state s, h^{max} returns the amount of forward iterations it took to compute the best supporter function in h^{FF}.

Definition 23: ($h^{U_{max}}$, $h^{U^2_{max}}$)
Similar to the definition of h^{max}, the values of $h^{U_{max}}$, $h^{U^2_{max}}$ can be set to the amount of forward iterations in $h^{U_{ff}}$, $h^{U^2_{ff}}$ respectively.

Definition 24: (Greedy plan extraction)
Recall how the plan extraction in te unary relaxation is defined. (Algorithm 4.) The additional instantiation are chosen via the $first$ function. Here I introduce an alternative: When it is possible to choose an instantiation s.t. some atom in the open list is reached, choose that. (Otherwise fall back to the $first$ function.) This should generally extract smaller plans.

Static unrelaxation:
In total the quadratic unrelaxation ($h^{U^2_{ff}}$) solves 27 more instances than the original configuration ($h^{U_{ff}}$) within the IPC benchmarks. Also the heuristic estimates seem to be a little higher, but still lower than h^{FF}. The highest difference can be found in an instance of pathways. (88 vs. 195, see figure 9.) In figure 6 of the appendix, you can see that $h^{U^2_{ff}}$has more expansions per time. The reason for this is that in $h^{U^2_{ff}}$the heuristic values (and so extracted plans) seem to shrink faster. This intuition should be really clear from the previous Logistics example in figure 4.3, where the h-value of $h^{U_{ff}}$ does not change when driving around, however for $h^{U^2_{ff}}$it does. The noise within the total

expansions scatterplot is expected, since there are still many pitfalls for both heuristics so tie breaking for such states matters a lot. Overall it seems reasonable to assume that $h^{U^2_{ff}}$is a slightly more qualitative heuristic than $h^{U_{ff}}$.

Split Predicate Transformation:
In earlier benchmarks, without the counter based version and with other successor generators, the Split Predicate Transformation showed a significant positive effect on both coverage and expansion. Now, however, this effect seems negligible. In total $h^{U^2_{ff}}$solves 6 more instances with the transformation, than without it. However, there are also instances, e.g. 3 in Sokoban, that are not solved anymore. Also in figure 1 of the appendix, both configurations seem nearly indistinguishable. In the plot for the initial heuristic values between $0-10$ there seems to be a little difference, but this could easily be the reason of a new predicate order through the newly introduced predicates. Since the transformation is really fast $\leq$ 20 seconds across all tasks, it should not have a negative impact to use the transformation, but it also does not seem like there is a need to use it. Note that this transformation may be not that effective, because it was already intuitively done by the task creator. E.g. the predicate *in* of the logistics task could have also be represented as *at*.

Partial Instantiation:
$pinst(spl(h^{U^2_{ff}}))$ initially performs multiple partial instantiations. Namely for each $(p, o, i) \in \{(p, \vec{o}_1, 1) \mid p(\vec{o}) \in \mathcal{G}\}$. Intuitively this means the first position of all goal atoms is instantiated. It is not surprising that this works well within the IPC domains, since these domains can be grounded. Note however, that in reality this is not entirely true since my implementation is really naive. E.g. in Organic Synthesis or Visitall the run time overhead of the partial instantiation is so big that less instances are solved. In Organic Synthesis and Organic Synthesis-split this even means that no instance will be solved anymore. Still, $pinst(spl(h^{U^2_{ff}}))$ solves 149 more instances than $spl(h^{U^2_{ff}})$. This show that even just such a partially grounded representation already provides a lot more information about the task that can be exploited.

Quality of the heuristic value computation
Still, h^{FF} solves 438 more instances than $pinst(spl(h^{U^2_{ff}}))$. Here I want to analyze different parts of the heuristic computation to get an idea what potential reasons for this could be. I will start by questioning the plan extraction. For this I compare standard $h^{U^2_{ff}}$ to $h^{U^2_{ff}}$with the greedy plan extraction ($greedy(h^{U^2_{ff}})$) in figure 7 of the appendix. You can see that this way the plans are indeed shorter, since the initial heuristic values are (almost strictly) smaller. However the amount of total expansions increases and the coverage decreases. (177 instances solved less in total.) So the usual plan extraction is at least better than the greedy plan extraction. Note that this potentially means that it is not desirable to compute h^{U*}.

To get a better idea about how useful the plan extraction is, I compare $h^{U_{ff}}$ to $h^{U_{max}}$ and $h^{U^2_{max}}$ to $h^{U^2_{ff}}$ in figure 4 and 5 of the appendix respectively. From both figures, it is very clear that the respective plan extractions outperform the simple forward iterations. This is also confirmed by the coverage difference. (725 vs. 476 and 752 vs. 482) With this it can not be concluded that this a good plan extraction in a sense of that alternatives wouldn't perform better, but at least this shows that the plan extraction makes a significant difference.

Coverage	BFS	GC	$h^{U_{ff}}$	$h^{U^2_{ff}}$	$spl(h^{U^2_{ff}})$	h^{FF}	GC$_{FD}$
genome-edit-distance (156)	20	**156**	25	26	24	62	**156**
genome-edit-distance-split (156)	18	**156**	19	20	20	35	**156**
organic-synthesis-alkene (18)	16	16	14	14	**17**	15	15
organic-synthesis-MIT (18)	**14**	**14**	12	11	13	2	2
organic-synthesis-original (20)	3	**6**	1	1	2	1	1
pipesworld-tankage-nosplit (50)	10	**21**	10	9	9	13	16
Sum (41)	81	**369**	81	81	85	128	346

Figure 5.2: Coverage table for the hard to ground domains.

As a last measure I compare $h^{U^2_{max}}$ to h^{max} and $spl(h^{U^2_{ff}})$, $pinst(spl(h^{U^2_{ff}}))$ to h^{FF} in figure 8. Here it seems like the difference in initial heuristic values decreases from left to right. Meaning that $spl(h^{U^2_{ff}})$ and $pinst(spl(h^{U^2_{ff}}))$ are closer to h^{FF} than $h^{U^2_{max}}$ is to h^{max}. My suspicion is that the reason for this phenomenon is that the heuristic value increases a lot by extracting a lot of actions that differ in only one split action. It would be interesting to return the amount of extracted split actions as heuristic value and see if this is a even more qualitative heuristic.

Hard to ground benchmarks:
Figure 5.2 shows the hard to ground benchmarks the Power Lifted Framework was initially tested on. The coverage scores of my tested approaches are comparably low, however this is expected, since h^{FF} also has comparably low coverage scores compared to GC within FD. Note however, that in organic-synthesis-alkene $spl(h^{U^2_{ff}})$scores the highest coverage score and so solves all instances of this domain except one. Predicate splitting seems to work well within Organic Synthesis in general. In all organic synthesis domains, $spl(h^{U^2_{ff}})$ solves more instances than h^{FF}.

5.3 A new hard to ground domain:

Visitall is a domain where a player has to visit all fields in a $n \times n$ grid from his starting position. A field is considered to be visited when the player walked on it once. The player can only walk to adjacent fields in the grid.

Here I introduce a set of domains, called visit-a-d that is closely related to visitall. a stand for some amount of fields that shall be visited and d for a dimension, meaning that the player moves in across positions in $\{1, ..., n\}^d$. The player can move from one position to another when the positions differ in only one position and in this position the difference of the elements is one. The goal is that the player visit a pre selected fields. This means visitall = visit-n^2-2.

Example 13: (visit-a-4)
Assume the player would be at position (1,2,7,1) of a visit-a-4 task. Then the player can move to the following positions:

- (1,1,7,1)
- (2,2,7,1)
- (1,2,8,1)
- (1,2,6,1)
- (1,3,7,1)
- (1,2,7,2)

I believe that doing further research with this domain will show strengths of the unary relaxation that I was not able to capture so far. And also addresses the following problems with recent lifted benchmarks:

- In the hard to ground domains above, there exists no predicate with an arity ≥ 3. As the arity of predicates grows, the amount of potential state grows. In the other domains this growth corresponds to the amount of transitions between states, not the amounts of states. With visit-a-d tasks there exists the option to scale the arity of predicates to any amount. In my opinion the rising amount of states is a really interesting problem to tackle with the lifted planning framework. E.g. consider a visit-a-4 task with a ridiculously high n, like $n = 10^9$. Then no planner would be able to compute a plan to go from $(1, 1, 1, 1)$ to (n, n, n, n), since just printing it would already take too long. However, reaching $(1, 1, 1, n)$ could possibly work, since then only $c * n$ states need to be computed. Maybe 10^9 is still too much, but the general idea should be understandable.

- In addition to the not too big state spaces, most current hard to ground task feature rather many goals. This means Goal Counting is expected to perform well. However, when this would not be the case, Goal Counting does not work well anymore and thus it will be a lot harder compute estimates. This is exactly the case in visit-a-d tasks. They can be designed to contain a small amount of goals in an enormous state space. I believe that in such a setting the unary relaxation will show to be more successful.

Chapter 6

Conclusion

With the help of the unary relaxation and the thereto related heuristics I was able to solve multiple tasks that the Power Lifted Planner was not able to solve so far. However, judging by the coverage in the IPC domains, goal counting is still has an edge over the here introduced benchmarks.

Even tough I described a potential way of further improving the heuristic with partial instantion, so far, I was not able to find a promising partial instantiation configuration. But I still believe that this has great potential of improving the heuristic. In order to do this, finding well working parameters is mandatory. Maybe one should start by doing this for some concrete domains and only generalize the approach from there.

Partial instantiation could also be extended even further. Instead of introducing a new predicate that represents one instantiation, the new predicate could also represent multiple instantiations. I.e. it will be only captured if this predicate is fulfilled for at least one of the instantiations. This would be similar to already existing work where equivalence classes are introduced to minimize the amount of atoms that need be computed. **??** The difference however is, that in the related work this equivalence classes are used to search in a smaller state space in a grounded fashion. The idea here would be to use the information of the equivalence classes to find proper parameters for such a partial instantiation and then continue using the unary relaxation.

Additionally the unary relaxation is not bound to a h^{FF} like plan extraction. E.g. one could try an approach similar to h^{lm-cut} [7].

The most important remaining question in my opinion, is how accurate the benchmarks with IPC benchmarks are with respect to hard to ground domains. The problem is, that currently it is really hard to benchmark the quality of lifted heuristics. IPC domains offer a wide variety, but are not hard to ground. Whereas there exist hard to ground domains, but these are really few and are lacking potentially important lifted structures. In my opinion it is crucial to invent new domains that are hard to ground and introduce different challenges for lifted heuristics for a further research of lifted heuristics.

Appendices

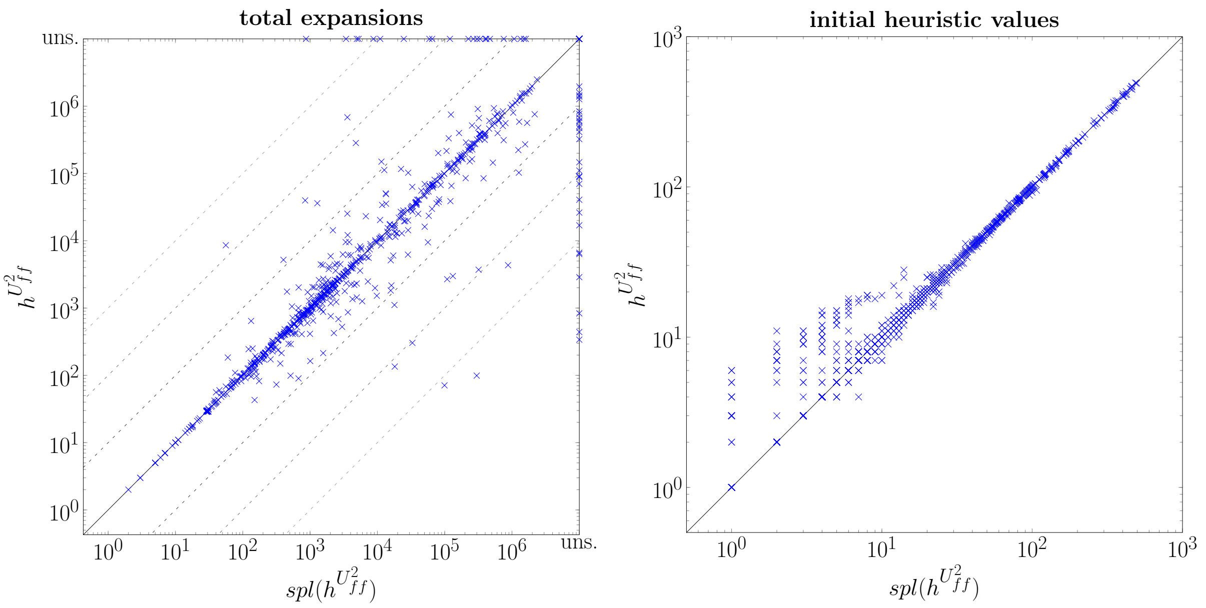
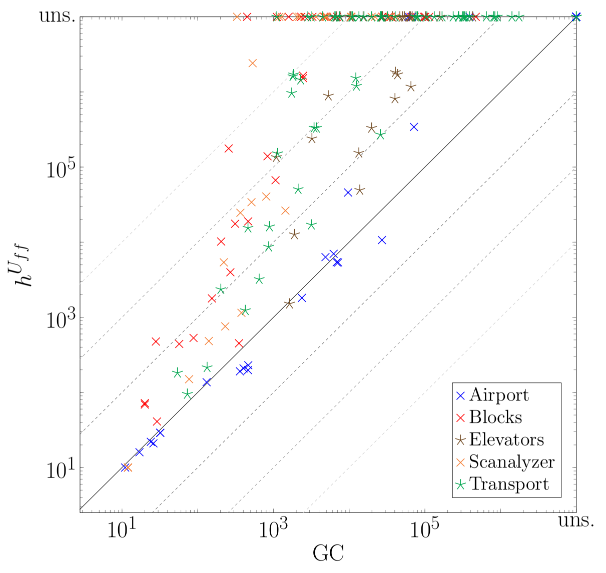


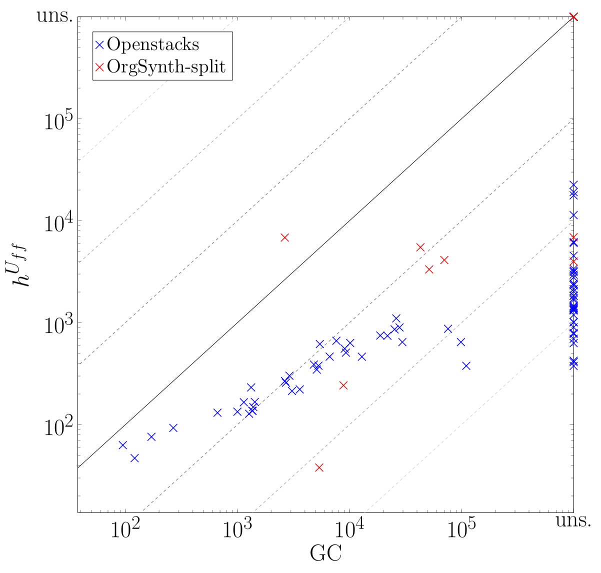
Figure 1: The comparison of expansions and initial heuristic values for $spl(h^{U^2_{ff}})$ and $h^{U^2_{ff}}$.

Coverage	BFS	GC	$h^{U_{max}}$	$h^{U^2_{max}}$	$h^{U_{ff}}$	greedy($h^{U^2_{ff}}$)	$h^{U^2_{ff}}$	$spl(h^{U^2_{ff}})$	$pinst(spl(h^{U^2_{ff}}))$	h^{max}	h^{ff}
Agricola (20)	0	0	5	**20**	4	0	**20**	**20**	**20**	**20**	**20**
Airport (50)	16	22	22	22	19	20	19	19	19	31	**36**
Barman (34)	0	18	0	0	4	0	4	4	16	11	**34**
Blocks (35)	15	**35**	21	21	17	18	17	17	**35**	**35**	**35**
Childsnack (20)	0	0	0	0	3	0	0	2	0	1	**7**
DataNetwork (20)	9	15	15	15	14	14	16	15	**19**	18	16
Depot (22)	2	11	3	3	3	3	4	4	9	9	**16**
DriverLog (20)	5	15	5	6	5	2	5	5	15	14	**18**
Elevators (30)	7	**30**	11	12	12	10	12	12	28	20	**30**
Floortile (40)	0	2	0	2	0	0	2	2	2	**31**	20
Freecell (80)	13	30	16	16	42	11	43	43	42	**80**	79
GED (20)	15	**20**	16	17	17	3	17	16	**20**	**20**	**20**
Grid (5)	1	2	0	0	0	0	0	0	**4**	1	**4**
Gripper (20)	6	**20**	6	6	6	6	6	6	**20**	8	**20**
Hiking (20)	8	8	9	9	9	9	9	9	14	**20**	**20**
Logistics (63)	11	30	11	11	11	11	11	11	23	27	**57**
Miconic (150)	40	**150**	62	51	134	135	**150**	**150**	**150**	66	**150**
Movie (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
Mprime (35)	14	18	1	1	1	1	1	1	22	24	**32**
Mystery (19)	11	13	2	2	2	2	2	2	15	16	**17**
Nomystery (20)	6	9	6	6	6	6	6	6	10	10	**15**
Openstacks (80)	12	37	14	14	**77**	76	**77**	76	76	15	36
OrgSynth (20)	10	12	12	12	12	9	12	**14**	0	7	7
OrgSynth-split (20)	6	6	8	8	8	3	8	8	0	16	**20**
Parcprinter (30)	7	14	8	7	8	7	9	7	12	17	**23**
Parking (40)	0	2	0	0	0	0	0	0	0	0	**40**
Pathways (30)	3	5	4	4	5	7	9	9	7	5	**10**
Pegsol (36)	31	**36**	32	31	35	1	35	35	16	**36**	**36**
Pipes-notank (50)	11	**32**	6	6	6	6	6	6	16	20	30
Pipes-tank (50)	6	18	6	6	7	0	6	6	8	14	**23**
PNetAlignment (20)	0	0	**20**	**20**	**20**	**20**	**20**	**20**	0	**20**	**20**
PSR (50)	39	46	42	42	46	46	46	46	46	**50**	**50**
Rovers (40)	4	15	6	6	17	13	17	17	19	9	**26**
Satellite (36)	2	6	3	3	4	3	4	6	15	6	**28**
Scanalyzer (30)	6	**30**	9	9	13	12	13	13	14	15	28
Snake (20)	1	5	6	6	6	6	6	6	0	12	**16**
Sokoban (30)	9	14	9	11	15	15	15	13	12	**28**	**28**
Termes (20)	3	15	5	1	2	1	3	2	9	15	**19**
Tetris (17)	5	**17**	3	4	15	1	14	13	2	13	15
Tidybot (40)	1	26	1	1	1	2	1	1	1	29	**37**
TPP (30)	5	10	5	5	6	6	5	6	16	9	**23**
Transport (60)	13	58	13	13	21	13	21	25	**59**	33	51
Trucks (30)	1	2	3	3	9	7	6	7	7	10	**14**
VisitAll (40)	7	**40**	7	7	**40**	29	33	36	19	13	39
Woodworking (30)	5	27	7	7	7	5	6	6	26	22	**30**
Zenotravel (20)	5	13	6	6	6	6	6	6	14	10	**20**
Sum (1622)	401	964	476	482	725	575	752	758	907	916	**1345**

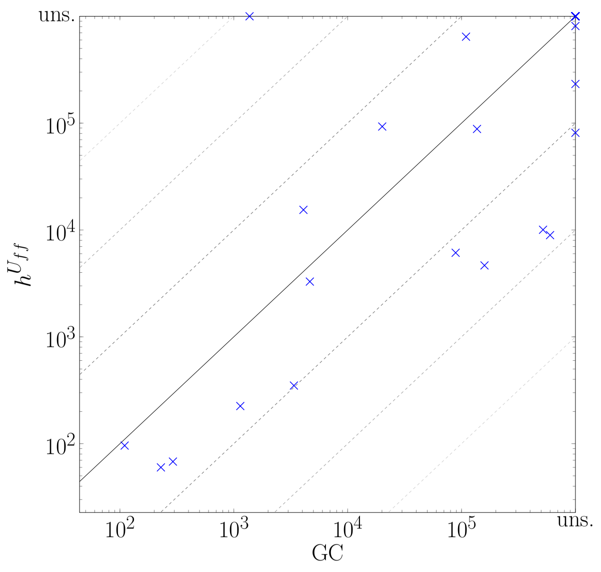
Figure 2: Coverage table for almost all run configurations.



(a) An example for domains where the expansions of GC tend to be generally lower.



(b) An example for domains where the expansions of $h^{U_{ff}}$ tend to be generally lower.



(c) The expansions for all Rovers instances.

Figure 3: In the subfigures different settings for relation of expansions between GC and $h^{U_{ff}}$ are illustrated.

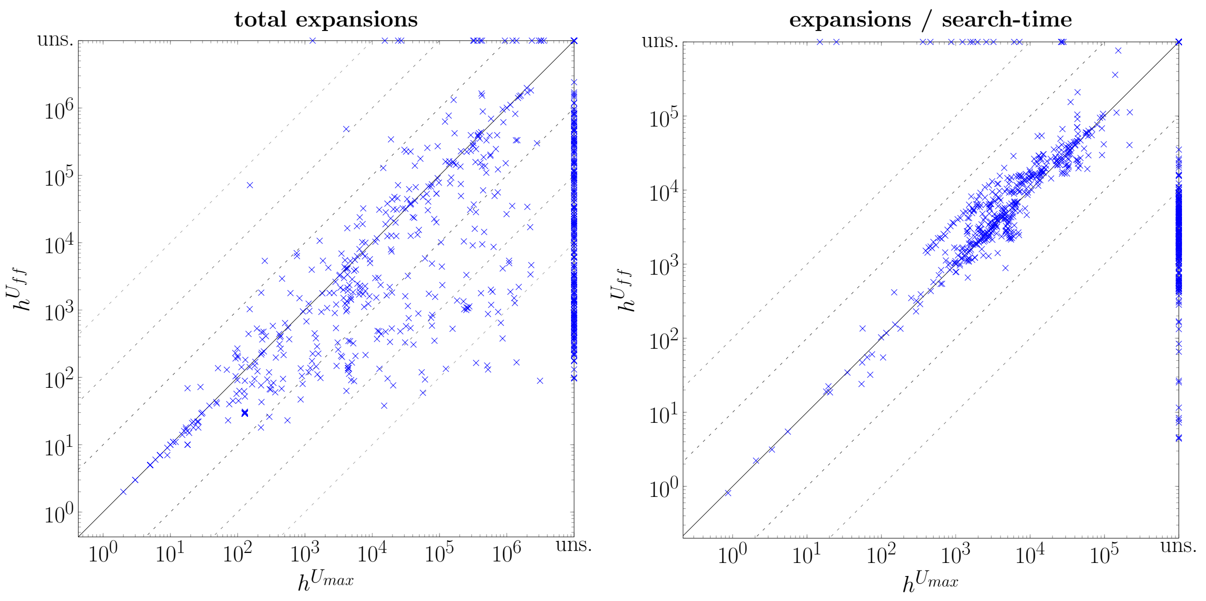


Figure 4: The comparison of expansion and expansions per search-time for $h^{U_{ff}}$ and $h^{U_{max}}$.

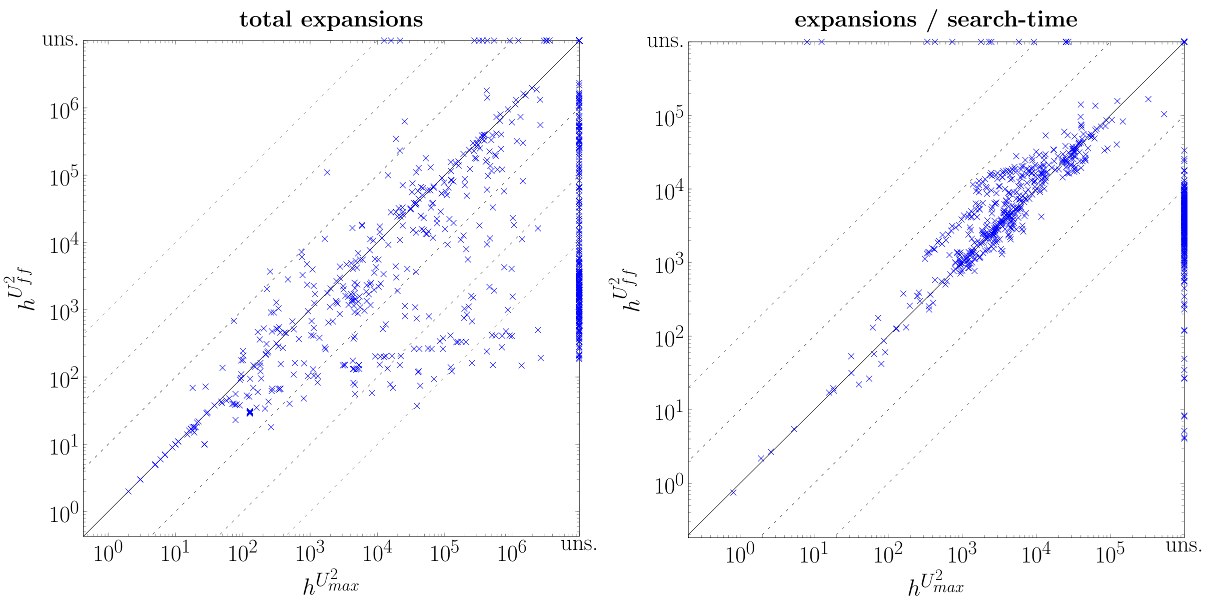


Figure 5: The comparison of expansion and expansions per search-time for $h^{U^2_{ff}}$ and $h^{U^2_{max}}$.

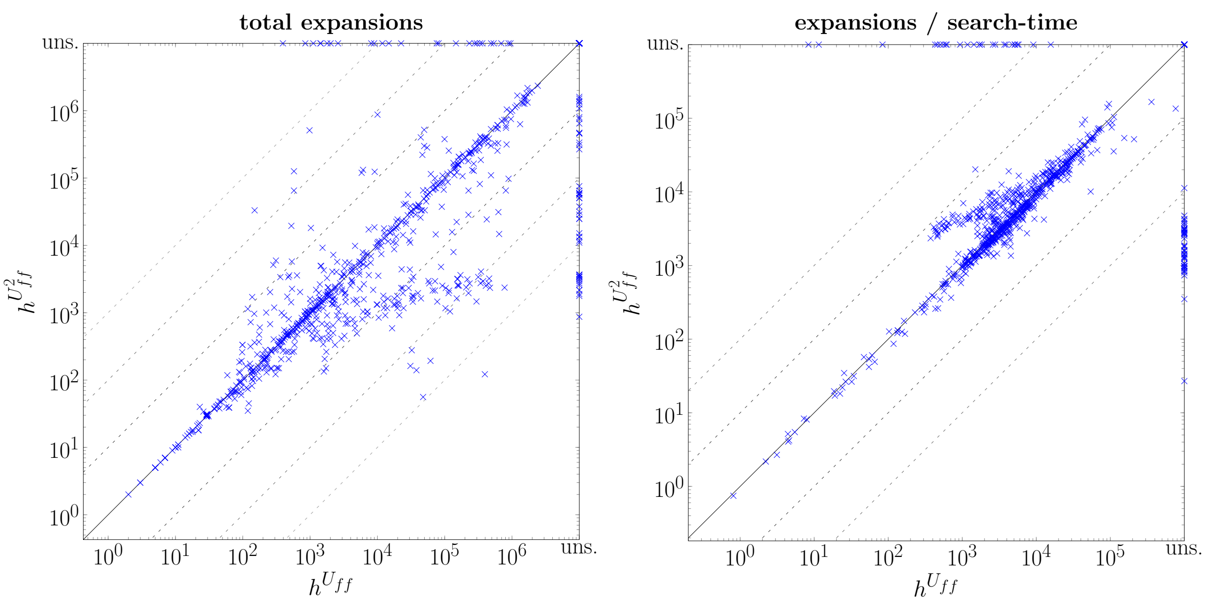


Figure 6: The comparison of expansion and expansions per search-time for $h^{U^2_{ff}}$ and $h^{U_{ff}}$.

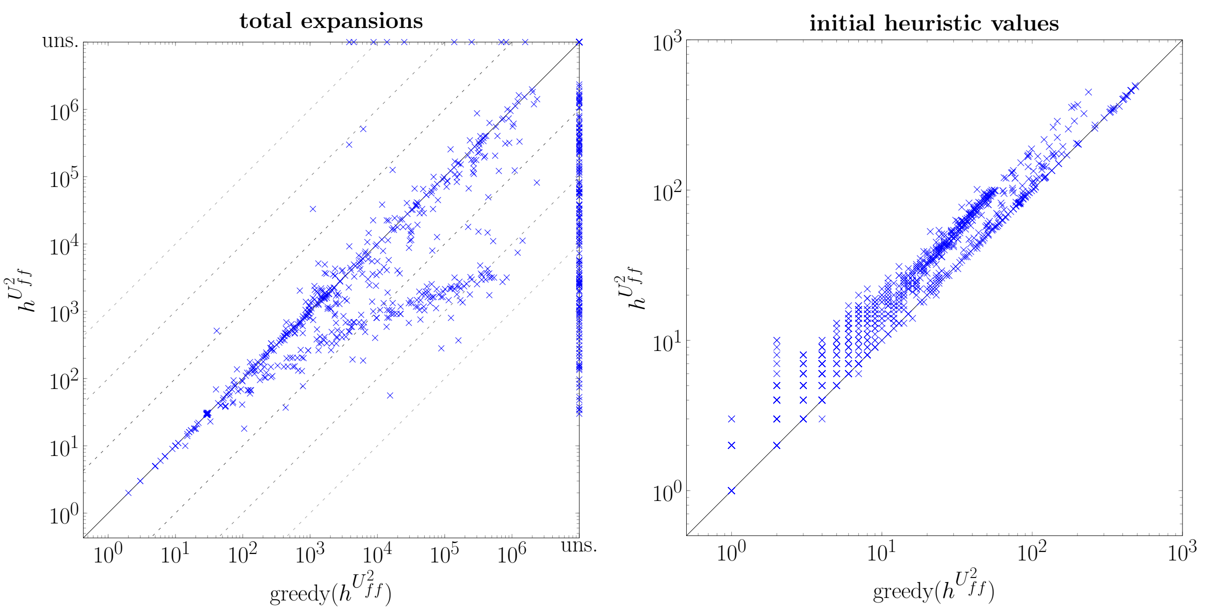
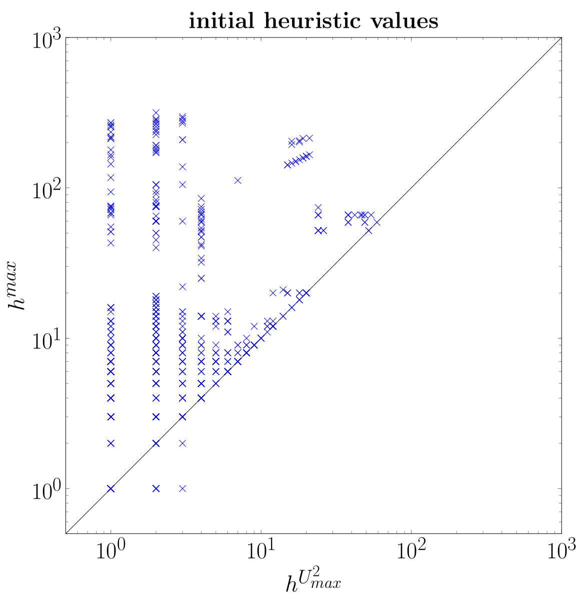
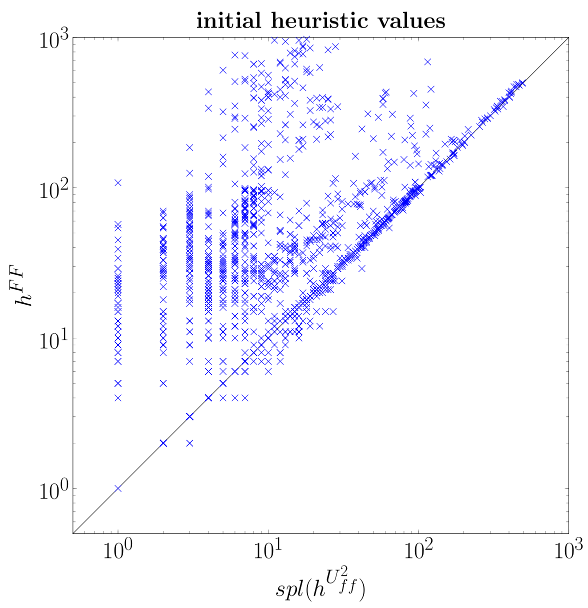


Figure 7: The comparison of expansions and initial for the classic vs. greedy approach of the plan exxtraction.





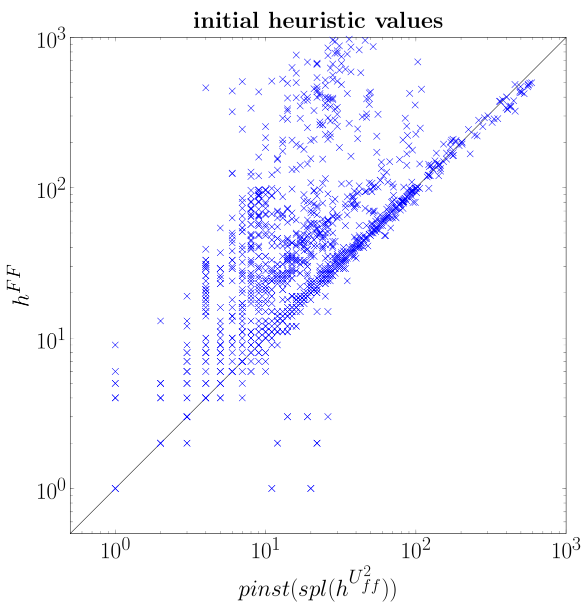


Figure 8: Comparison of initial heuristic values in delete relaxation vs. the unary relaxation.

Initial h value	GC	$h^{U_{max}}$	$h^{U^2_{max}}$	$h^{U_{ff}}$	$h^{U^2_{ff}}$	$spl(h^{U^2_{ff}})$	$pinst(spl(h^{U^2_{ff}}))$	h^{FF}
Agricola	2	9	21	10	29	29	30	**398**
Airport	16	59	59	121	121	121	109	**687**
Barman	9	4	4	26	30	29	24	**58**
Blocks	17	3	3	15	16	15	32	**33**
Childsnack	16	3	3	18	20	**21**	**21**	**21**
DataNetwork	7	3	4	15	17	17	17	**208**
Depot	18	2	2	12	12	12	35	**87**
DriverLog	32	1	2	3	6	6	71	**106**
Elevators	8	3	3	10	10	10	21	**96**
Floortile	37	2	6	41	63	63	67	**172**
Freecell	5	2	2	8	9	9	13	**97**
GED	11	1	1	3	3	3	**26**	3
Grid	8	5	12	16	47	47	**73**	64
Gripper	43	2	2	3	3	3	**85**	**85**
Hiking	3	3	9	3	15	15	22	**47**
Logistics	45	1	1	22	22	22	44	**296**
Miconic	31	3	3	61	**100**	**100**	**100**	**100**
Movie	**8**	1	1	7	7	7	7	7
Mprime	3	1	1	1	1	1	8	**14**
Mystery	3	1	1	1	1	1	8	**14**
Nomystery	13	1	2	4	4	11	**33**	**33**
Openstacks	101	4	4	**403**	402	**403**	402	401
OrgSynth (20,13,13,13,13,13,0,7)	**21**	1	1	2	2	5	0	4
OrgSynth-split (20,13,13,13,13,13,13,20)	21	7	7	14	14	25	42	**902**
Parcprinter	32	8	8	46	46	46	56	**3763415**
Parking	23	3	3	27	28	28	**48**	**48**
Pathways	41	4	12	88	195	**196**	194	191
Pegsol	**34**	1	3	32	31	31	24	24
Pipes-notank	9	3	3	6	6	6	28	**55**
Pipes-tank	9	3	3	6	6	7	27	**56**
PNetAlignment	4	403	403	493	496	492	**590**	499
PSR	**8**	3	3	7	7	7	7	7
Rovers	70	3	4	168	171	171	177	**210**
Satellite	232	3	3	444	449	451	457	**486**
Scanalyzer (30,30,30,30,30,30,24,30)	25	1	1	18	18	18	35	**79**
Snake	6	0	0	0	0	0	0	**23**
Sokoban	13	1	4	12	12	15	15	**59**
Termes	8	2	8	4	8	8	29	**44**
Tetris (17,17,17,17,17,17,2,17)	15	2	3	15	15	15	5	**28**
Tidybot	5	1	6	4	10	12	11	**36**
TPP	21	4	9	9	9	9	64	**127**
Transport	12	1	3	10	14	28	52	**1344**
Trucks	21	4	4	**67**	66	**67**	66	66
VisitAll	324	1	18	323	323	323	**458**	323
Woodworking	42	3	3	17	17	19	46	**1030**
Zenotravel	26	1	1	3	3	3	61	**68**
Max	324	403	403	493	496	492	590	**3763415**

Figure 9: Maximal inital heuristic value table.

Bibliography

[1] Malte Helmert Augusto B. Corrêa, Florian Pommerening and Guillem Francès. Lifted successor generation using query optimization techniques. AAAI Press, 2020.

[2] Christer Bäckström and Bernhard Nebel. Complexity results for SAS$^+$ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[3] Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Francès. Code from the paper ”lifted successor generation using query optimization techniques”. Feb 2020.

[4] Richard E. Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[5] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[6] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173:503–535, 2009.

[7] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 2009.

[8] Malte Helmert and Silvia Richter. Fast downward – making use of causal dependencies in the problem representation. In Stefan Edelkamp, Jörg Hoffmann, Michael Littman, and Håkan Younes, editors, *Proceedings of the 4th International Planning Competition*, Whistler, BC, Canada, Jun 2004. JPL.

[9] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research - JAIR*, 14, 05 2001.

[10] Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski. Modeling organic chemistry and planning organic synthesis. In Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors, *GCAI 2015. Global Conference on Artificial Intelligence*, volume 36 of *EPiC Series in Computing*, pages 176–195. EasyChair, 2015.

[11] Rami Matloob and Mikhail Soutchanski. Exploring organic synthesis with stateofthe-art planning techniques. 2016.

[12] Drew McDermott et al. *The PDDL Planning Domain Definition Language.* The AIPS-98 Planning Competition Comitee, 1998.

[13] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward lab. https://doi.org/10.5281/zenodo.790461, 2017.

[14] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, page 82–94. VLDB Endowment, 1981.