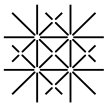


University of Basel

Unrolling of Negative Axioms in Delete Relaxation Heuristics

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Research Group Artificial Intelligence
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Gabriele Röger

Patrick Weber
patrick01.weber@stud.unibas.ch
2022-051-924

25.01.2026

Acknowledgments

I would like to thank my supervisor Dr. Gabriele Röger for her great feedback, guidance and explanations of the new topics I encountered during this work. I would also like to thank Prof. Dr. Malte Helmert for giving me the opportunity to write my Bachelor's thesis in the Artificial Intelligence Research Group.

Abstract

Classical planning is a fundamental area of artificial intelligence, which focuses on finding an action sequence from an initial state to a goal state. Many planning systems make use of axioms, which define facts that are not directly changed by actions but model logical rules. In this thesis we introduce unrolling, a method to transform cyclic dependencies into a provably exact acyclic representation, to compute negative axioms for delete relaxation heuristics.

We compared unrolling to the pre-existing approximation methods by conducting experiments on specifically chosen benchmark suites that contain such cyclic dependencies. Our results show that unrolling is an effective method to handle cyclic dependencies and improves the heuristic search. However, there is a trade-off that comes with a substantial increase in memory usage and preprocessing time. We conclude that unrolling demonstrates its best performance for domains that have small cycles.

Table of Contents

1 Introduction

Classical planning is an essential part of Artificial Intelligence that involves finding a sequence of actions from an initial state to a goal state. This is done by performing a search on predefined actions. Which action should be taken next is decided by heuristics, which assign a value to each state that estimate how expensive it is to reach the goal from that state. The heuristics we use are delete relaxation heuristics, which only take into account the positive effects of an action and ignores the negative effects.

Many planing systems make use of axioms, which define facts that are not directly changed by actions but model logical rules. Axioms derive whether some variable is true or not based on other variables. This creates dependencies between the variables. These dependencies can be modeled as a graph. There may be cyclic dependencies, which can be found by detecting cycles in this graph. This is a rather rare occurrence but has to be treated with special care.

The current implementation of Fast Downward supports solving problems with these axioms, including those cyclic dependencies. Since we use delete relaxation heuristics, we need to perform a task transformation first to calculate the negative axioms that this heuristic needs. However, there are currently only approximations implemented, since the normal way for creating these negative axioms for cycles does not work. There have been talks about how to implement the exact calculation of these cycles, which in theory creates a huge calculation and memory overhead. The number of axioms needed to represent a cycle exactly is polynomial to the cycle size. Therefore, this implementation has not been done yet.

The goal of this thesis is to implement the exact calculation of cycles with unrolling. Unrolling is a method which essentially transforms cyclic dependencies into an acyclic representation. With the new acyclic representation, we can exactly calculate the negative axioms. The exact effects of using unrolling for this case are unknown and will be analyzed in this thesis and tested against the implementations that use the approximations.

In Chapter 2, we will introduce the necessary terminology and methods. In Chapter 3.1, we will talk about what unrolling is and how it can be implemented. Next, in Chapter 4, we will introduce the other methods that have been used to calculate default value axioms until now. Then, in Chapter 5, we will perform experiments to compare the unrolling implementation to the already existing approximate implementation and discuss the results. And lastly, in Chapter 6, we will recap our findings and talk about future work that could be done.

2
Background

In this chapter we will cover the necessary terminology and methods needed to understand our main implementation.

2.1 Axioms

Axioms were first introduced in PDDL 2.2 [Edelkamp and Hoffmann, 2004]. We change the original definition so that the representation is consistent with the one used in the Fast Downward planner, which uses finite-domain representation [Helmert, 2009].
We define the variables in the context of a finite-domain representation. Each variable v has a finite domain $dom(v)$. For ease of presentation, we will use the terms *derived variable* and *state variable*.

Definition 1 (Derived Variable)**.** *A* derived variable v^d *is a variable that has boolean value* $\{F, T\}$*. Every derived variable is initialized as* F*. We further define the set of all derived variables as* $\mathcal{V}^d$*.*

Definition 2 (State Variable)**.** *A* state variable v^s *is a variable that has a finite domain* D_v*. We further define the set of all state variables as* $\mathcal{V}^s$*.*

We define the set of all variables as $\mathcal{V} = \mathcal{V}^d \cup \mathcal{V}^s$.
Now that we defined the variables and their domains, we can define what an axiom is.

Definition 3 (Axiom)**.** *An* axiom *is a rule* $a := (h = T \leftarrow \varphi)$*, where* h *is called the head of the axiom and* φ *the body. The* head h *of an axiom is defined as* $h \in \mathcal{V}^d$*. The* body φ *of an axiom is a conjunction of atoms* $c := (v = x)$ *where* $v \in \mathcal{V}$ *and* $x \in dom(v)$*. We further define the set of all axioms as* $\mathcal{A}$*.*

Definition 4 (Negative Axiom)**.** *A* negative axiom *is an axiom of form* $h = F \leftarrow \varphi$*.*

We define $c \in \varphi$ as an atom that is part of the conjunction that builds φ. The body of an axiom $a := (h = T \leftarrow \varphi)$ is satisfied iff every atom $c \in \varphi$ evaluates to true. If the body of an axiom $(h = T \leftarrow \varphi) \in \mathcal{A}$ is satisfied, we set $h = T$. Respectively, if the body of a negative axiom $(h = F \leftarrow \varphi) \in \mathcal{A}$ is satisfied, we set $h = F$.

For ease of access, we define the variables of all atoms of a body φ as $vars(\varphi) = \{v \mid (v = x) \in \varphi\}$ and the value x needed to satisfy an atom $c := (v = x)$ as $val(c)$.

Consider as a running example Figure 2.1. The example shows an agent A, three rooms labeled as R_1, R_2 and R_3 and four doors labeled as d_0, d_1, d_2 and d_3. The goal of the agent is to open all the doors. To open a door d_i it needs a key k_i. To keep the example simple, the key k_i is always in room R_i and door d_i can only be opened from room R_i, which is denoted by the arrows in the graph. The agent can perform certain actions, such as opening door d_i with key k_i, passing an open door to enter a different room or pick up the key k_i in room R_i. The specific actions are not relevant for us, since we will only focus on the axioms, which derive facts that follow logical rules. At the start, the agent has key k_0, which makes $keyUsable0 = T$.

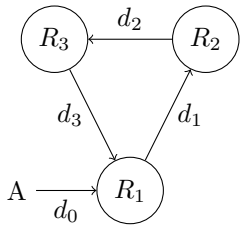


Figure 2.1: Example problem for an agent who has to open the doors for all rooms

We define the axioms for this problem in Eq. (1).

$$\begin{aligned}
roomReachable1 = T &\leftarrow doorOpenable0 = T \\
roomReachable1 = T &\leftarrow roomReachable3 = T \wedge doorOpenable3 = T \\
roomReachable2 = T &\leftarrow roomReachable1 = T \wedge doorOpenable1 = T \\
roomReachable3 = T &\leftarrow roomReachable2 = T \wedge doorOpenable2 = T \\
doorOpenable0 = T &\leftarrow keyUsable0 = T \\
doorOpenable1 = T &\leftarrow keyUsable1 = T \wedge roomReachable1 = T \\
doorOpenable2 = T &\leftarrow keyUsable2 = T \wedge roomReachable2 = T \\
doorOpenable3 = T &\leftarrow keyUsable3 = T \wedge roomReachable3 = T
\end{aligned} \tag{1}$$

As an example, consider variable *roomReachable2*. Since this variable is the head of an axiom, we know that it is a derived variable. Derived variables are initialized as false. This variable is set to true if all atoms of its body are satisfied, i.e. room R_2 is reachable if room R_1 is reachable and door d_1 can be opened.

2.1.1 Stratification

Axioms need an order in which they get evaluated, this is done with stratification.

Definition 5 (Stratification)**.** Stratification *is a layering function* $l(x) : x \to \mathbb{N}_0$*, where* $x \in \mathcal{V}^d$ *with rules:*

- *If* $(y = T) \in \varphi$ *with* $y \in \mathcal{V}^d$ *and* $(x = T \leftarrow \varphi) \in \mathcal{A}$*, then* $l(y) \leq l(x)$*.*
- *If* $(y = F) \in \varphi$ *with* $y \in \mathcal{V}^d$ *and* $(x = T \leftarrow \varphi) \in \mathcal{A}$*, then* $l(y) < l(x)$*.*

Since axioms do not change state variables, we define $l(x) = 0$ *for all* $x \in \mathcal{V}^s$*.*

Definition 6 (Stratifiable)**.** *A set of variables* V *is* stratifiable *if there exists a stratification for the set of all variables* $\mathcal{V}$*.*

Just like variables, we also create a stratification for axioms. To keep things simple, we say that $l(a) = l(x)$ where $a := (x = T \leftarrow \varphi) \in \mathcal{A}$.

2.1.2 Axiom Evaluation

We define the evaluation of axioms using a staged approach based on [Grundke and Röger, 2025]. This method shows how one layer of axioms is evaluated. We define $\Pi \subseteq \mathcal{A}$ as an arbitrary axiom layer and S as a partial assignment. We define $S(v) = x$ for every $v \in \mathcal{V}$ with $x \in dom(v)$. The assignment S contains all variables that were evaluated in previous layers, whose values are fixed, and the set of derived variables that will be evaluated from this layer $\Pi_V = \{v \in \mathcal{V}^d \mid (v = T \leftarrow \varphi) \in \Pi\}$, which are initialized as false.

Algorithm 1 Extension for an axiom layer in stages (Adapted from [Grundke and Röger, 2025])

```
1: function ExtendAxiomLayerInStages(axiom layer Π, truth assignment S)
2:     for j ∈ 0, ... do
3:         S_j = copy of S
4:         while there exists a rule (h = T ← φ) ∈ Π with S_j ⊨ φ and S(h) = F do
5:             Choose such an h = T ← φ and set S(h) = T
6:         if S = S_j then return
```

The algorithm creates a sequence $S_0, \ldots, S_n$ where S_k denotes the assignment at step $k \in \{0, \ldots, n\}$ and $n = |\Pi_V|$.

Definition 7 (Fixed Point [Röger and Grundke, 2024])**.** *A fixed point for an axiom layer* Π *is reached when* $S_k = S_{k+1}$ *in the sequence of the axiom evaluation* $S_0, \ldots, S_n$*. We further denote the assignment of the fixed point as* S_n*.*

The fixed point S_n is exactly the truth assignment that is reached after running algorithm ExtendAxiomLayerInStages on an arbitrary axiom layer Π with initial truth assignment S.

2.1.3 Dependencies

The axioms induce dependencies between the derived variables. We define these as default and non-default dependencies, which describe how a derived variable $y \in vars(\varphi)$ occurs in the body of an axiom $(x = T \leftarrow \varphi) \in \mathcal{A}$.

Definition 8 (Default Dependency)**.** *Variable* x *has a* default dependency *on variable* y *if there exists an axiom* $(x = T \leftarrow \varphi) \in \mathcal{A}$ *with* $(y = F) \in \varphi$*.*

Definition 9 (Non-Default Dependency)**.** *Variable x has a* non-default dependency *on variable y if there exists an axiom $(x = T \leftarrow \varphi) \in \mathcal{A}$ with $(y = T) \in \varphi$.*

These dependencies can be represented as a graph. We ignore the default dependencies, since they require a derived variable x to be $x = F$, which is how we initialized them, and therefore the default dependencies between derived variables are not relevant.

Definition 10 (Dependency Graph)**.** *The non-default dependencies form a directed graph $G = (V, E)$, called the* dependency graph*, with vertices $V = \mathcal{V}^d$ and edges $E = \{(x, y) \mid y \in \mathcal{V}^d$ has a non-default dependency on $x \in \mathcal{V}^d\}$.*

The dependency graph describes which derived variables $vars(\varphi) \in \mathcal{V}^d$ need to be true in order to satisfy $h = T \leftarrow \varphi$ and set $h \in \mathcal{V}^d$ to true. The strongly connected components (further only called "SCCs") can be computed in this graph. They are used to finding cycles in the graph. We define those cycles in the dependency graph as cyclic dependencies. These cyclic dependencies are what we need to consider for unrolling.

We create the dependency graph from the example, where we abbreviate *roomReachableI* as R_i, *doorOpenableI* as D_i and *keyUsableI* as K_i. In Figure 2.2 we can see several cyclic dependencies, for example $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_1$. We will use that cycle as a running example throughout Chapter 3. Although, this is technically incorrect since we always take the biggest cycle into consideration, which would also include derived variables D_1, D_2 and D_3.

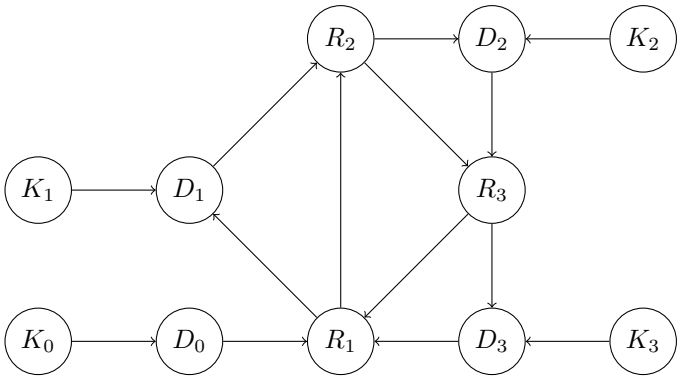


Figure 2.2: Dependency graph for axioms in Eq. (1)

3
Unrolling

In this chapter we will discuss unrolling as a method transform the cyclic dependencies of the dependency graph into an acyclic representation and prove that it is still an exact representation of the original cycle.
Our goal is to create negative axioms, which are for example needed for delete relaxation heuristics. They describe how a derived variable can achieve its initial value (which we defined as F).
We usually do not need axioms to access the initial values of variables, since we are only interested in the current value of a variable (if a derived variable is set to T it is no longer F). This is different for delete relaxation heuristics, since they do not remove any information from a variable (if a derived variable is set to T its old value F can still be used).
The normal way to create these negative axioms does not work for cycles. This is why we first need to create an acyclic representation of those cycles in order to compute the negative axioms for the derived variables that are contained in cycles.
We will talk about why we cannot compute the negative axioms for cyclic dependencies and how the implementation has been done until now in Chapter 4.

3.1 Concept

We will take a closer look at the cyclic dependencies of the dependency graph. Cycles are detected by computing the SCCs of the graph. If an SCC contains more than one variable, it contains a cycle. From now on, we will only consider SCCs with more than one variable. An SCC is a set $V \subseteq \mathcal{V}^d$, that describes a set of axioms $A_V = \{(h = T \leftarrow \varphi) \in \mathcal{A} \mid h \in V\}$ and induces a sequence of partial assignments $S_0, S_1, \ldots$ that we define as follows: Let S_0 be a partial assignment for every variable $v^p \in vars(\varphi) \setminus V$ that occurs in axioms $(h = T \leftarrow \varphi) \in A_V$ and for every variable $v \in V$. The values for the variables v^p depend on what has been derived in previous axiom layers and are therefore fixed. The values for

v are initialized as F. We define S_{k+1} as shown in Eq. (2)

$$S_{k+1}(v) = \begin{cases} S_k(v) & \text{if } v \notin V \\ T & \text{if } (v = T \leftarrow \varphi) \in A_V \text{ with } S_k \models \varphi \\ F & \text{otherwise} \end{cases} \tag{2}$$

Since the axioms are stratifiable, we know that there is no $(v = F) \in \varphi$ with $v \in V$ and $(h = T \leftarrow \varphi) \in A_V$. From this we know that the set of variables where $S_k(v) = T$ is monotone. Since V is finite, and we only set variables in S_k to true, the sequence $S_0, S_1, \ldots$ will eventually reach a fixed point where $S_k = S_{k+1}$. Since the set of variables that are true in the sequence is monotone, and we can set at most $n = |V|$ variables to true, the fixed point S_n will be reached at the latest after step n.

We now unroll the cycle of SCC V by using $n-1$ different copies of the axioms A_V. We assign each copy a timestamp $k \in \{0, \ldots, n-2\}$, so that we can differentiate the copied axioms. For each copy, we change the original variables from V to an unrolling variable.
Let $V^{\text{unroll}} = \{v^{(k)} \mid v \in V, k \in \{0, \ldots, n-2\}\}$ be the set of unrolling variables where $v^{(k)}$ describes a variable $v \in V$ at timestamp k. For a formula φ and $k \in \{0, \ldots, n-2\}$ we define $\varphi^{(k)}$ as the formula that results from replacing every $v \in \mathcal{V}^d$ with $v^{(k)}$. The numbers may seem a bit random, but the logic holds: We create $n-1$ different copies of the axioms, since they are of the form $v^{(k+1)} = T \leftarrow \varphi^{(k)}$ and we have n different timestamps. We need n different timestamps for k but since we map the last axiom back to the original variable $v = T \leftarrow \varphi^{(n-2)}$, we still have n different variables $(v^{(0)}, \ldots, v^{(n-2)}, v)$, which is exactly what we need.
Our goal is to create a new set of axioms A_V^{unroll} that exactly represents the original axioms A_V, which means that for the fixed point of the unrolling axioms S_n^{unroll} it holds that $S_n(v) = S_n^{\text{unroll}}(v)$ for all $v \in V$. The unrolling axioms are constructed as follows:

Definition 11 (Unrolling Axioms)**.** *The set of* unrolling axioms *for SCC V is defined as $A_V^{unroll} = A^P \cup A^D \cup A^I$ where*

- $A^P = \{v^{(k+1)} = T \leftarrow v^{(k)} = T \mid v \in V, k \in \{0, \ldots, n-3\}\} \cup \{v = T \leftarrow v^{(n-2)} = T \mid v \in V\}$ *is the set of* propagation axioms.
- $A^D = \{v^{(k+1)} = T \leftarrow \varphi^{(k)} \mid (v = T \leftarrow \varphi) \in A_V, k \in \{0, \ldots, n-3\}, vars(\varphi) \cap V \neq \emptyset\}$ $\cup \{v = T \leftarrow \varphi^{(n-2)} \mid (v = T \leftarrow \varphi) \in A_V, vars(\varphi) \cap V \neq \emptyset\}$ *is the set of* cycle-dependent axioms.
- $A^I = \{v^{(0)} = T \leftarrow \varphi \mid (v = T \leftarrow \varphi) \in A_V, vars(\varphi) \cap V = \emptyset\}$ *is the set of* cycle-independent axioms.

We take a look at our running example and create some example unrolling axioms for the cycle $R_1 \to R_2 \to R_3 \to R_1$ from Figure 2.2:

$$roomReachable2^{(2)} = T \leftarrow roomReachable1^{(1)} = T \tag{3}$$

$$roomReachable2^{(2)} = T \leftarrow roomReachable1^{(1)} = T \wedge doorOpenable1^{(1)} = T \tag{4}$$

$$roomReachable1^{(0)} = T \leftarrow doorOpenable0 = T \tag{5}$$

Eq. (3) shows a propagation axiom, which describes that if room R_2 is reachable at $k = 1$, then it is also reachable at $k = 2$. We need propagation axioms to ensure that if a derived variable is true, it is also true in the next timestamp.
Eq. (4) shows a cycle-dependent axiom, which describes that if R_1 is reachable and door d_1 can be opened at $k = 1$, then room R_2 is reachable in $k = 2$. Cycle-dependent axioms set a derived variable to true if its body is satisfied from the partial assignment of the previous timestamp.
Eq. (5) shows a cycle-independent axiom, which describes that room R_0 is reachable at $k = 0$ if door d_0 can be opened. Since *doorOpenable0* is not a derived variable of the SCC we unrolled, it has to be set to true by an axiom of a previous layer. Therefore, we only need one axiom, which tells us if *roomReachable0* is true at $k = 0$ for that axiom. If it is true, then the propagation axiom ensures that it is also true for future timestamps.

We write this down as an algorithm that creates all unrolling axioms for SCC V. The input for algorithm UNROLLCYCLE is the SCC V it unrolls and the set of all axioms $\mathcal{A}$.

Algorithm 2 Unrolling

```
1:  function UNROLLCYCLE(SCC V, axioms A)
2:      n = |V|
3:      Create unrolling variables V^unroll = {v^(k) | v ∈ V, k ∈ {0, ..., n}}
4:      for t ∈ {0, ..., n − 2} do
5:          for var in V do
6:              for (var = T ← φ) ∈ A do
7:                  if v ∉ V for every v ∈ vars(φ) and t = 0 then
8:                      Create cycle-independent axiom var^(0) = T ← φ
9:                  else
10:                     Create cycle-dependent axiom var^(k+1) = T ← φ^(k)
11:         Create propagation axiom var^(k+1) = T ← var^(k) = T
```

Now that we have established all rules of unrolling, we go back to our cycle from Figure 2.2, unroll the cycle $R_1 \to R_2 \to R_3 \to R_1$, list the created unrolling axioms in Eq. (6) and show the dependency graph in Figure 3.1. Since the cycle has 3 derived variables we use $n = 3$. We assign a number to each axiom in squares which will be used later.

$$
\begin{aligned}
roomReachable1^{(0)} = T &\leftarrow doorOpenable0 = T \ [0] \\
roomReachable1^{(1)} = T &\leftarrow roomReachable3^{(0)} = T \wedge doorOpenable3 = T \ [1] \\
roomReachable1 = T &\leftarrow roomReachable3^{(1)} = T \wedge doorOpenable3 = T \ [2] \\
roomReachable2^{(1)} = T &\leftarrow roomReachable1^{(0)} = T \wedge doorOpenable1 = T \ [3] \\
roomReachable3 = T &\leftarrow roomReachable1^{(1)} = T \wedge doorOpenable1 = T \ [4] \\
roomReachable3^{(1)} = T &\leftarrow roomReachable2^{(0)} = T \wedge doorOpenable2 = T \ [5] \\
roomReachable3 = T &\leftarrow roomReachable2^{(1)} = T \wedge doorOpenable2 = T \ [6] \\
roomReachable1^{(1)} = T &\leftarrow roomReachable1^{(0)} = T \ [7] \\
roomReachable1 = T &\leftarrow roomReachable1^{(1)} = T \ [8] \\
roomReachable2^{(1)} = T &\leftarrow roomReachable2^{(0)} = T \ [9] \\
roomReachable2 = T &\leftarrow roomReachable2^{(1)} = T \ [10] \\
roomReachable3^{(1)} = T &\leftarrow roomReachable3^{(0)} = T \ [11] \\
roomReachable3 = T &\leftarrow roomReachable3^{(1)} = T \ [12]
\end{aligned}
\tag{6}
$$

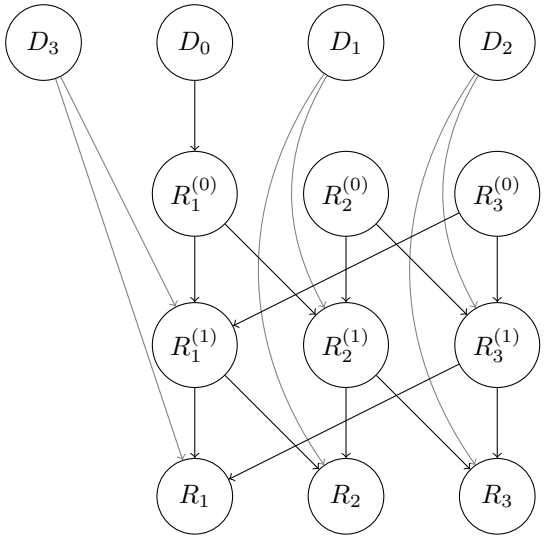


Figure 3.1: Dependency graph for axioms Eq. (6)

3.1.1 Stratification

It is important that stratifiability is preserved with the newly created unrolling axioms, since the creation of those would not be allowed otherwise. We can assume that the original set of axioms is stratifiable.

Something to note is that derived variables with cyclic dependencies have to be part of the same layer. We show this with a specific cycle from Figure 2.2. We have the cycle $R_1 \to R_2 \to R_3 \to R_1$, which means that we have a non-default dependency from R_2 on R_1, from R_3 on R_2 and so on. From our stratification rules, this means that $R_1 \leq R_2$

(since $R_1 = T$ appears in the body of an axiom $(R_2 = T \leftarrow \varphi) \in \mathcal{A}$), $R_2 \leq R_3$ and so on. If we combine these we get the formula $\cdots \leq R_1 \leq R_2 \leq R_3 \leq R_1 \leq \ldots$ since we can loop the cycle infinitely. From this we follow, that the only solution of this equation is $\cdots = R_1 = R_2 = R_3 = R_1 = \ldots$, which forces them to be in the same layer.

Theorem 1. *If variables V_L with $V \subseteq V_L$, where V is an SCC that needs to be unrolled, are stratifiable, then the extension of that layer $V_L^{unroll} = V_L \cup V^{unroll}$ is also stratifiable.*

Proof. The set of all variables $\mathcal{V}$ consists of several SCCs. These SCCs form a directed acyclic graph. Since the dependencies between the SCCs are acyclic, we can topologically sort them and assign them a temporary index i. Consider an arbitrary SCC with index i. The variables from this SCC only read the values from variables v^r with index $j \leq i$. They also only appear in the body of axioms with heads v^w with index $k \geq i$. Stratification forces variables into the same layer only when cyclic dependencies exist. This is the case when variables are both read from and written to by the same variables ($j \leq i \leq k$, which implies $j = i = k$). These are exactly the variables within the cycle itself. Therefore, there are no constraints that force V to share a layer with variables from another SCC, and we can assign each SCC to a distinct layer. Thus, we assume without loss of generality that V constitutes its own layer in this proof.
We consider the three types of different axioms to determine which dependencies the derived variables in V^{unroll} have:

1. *Cycle-independent axioms:* Their bodies do not contain any $v \in V$. Those variables are not part of V_L and reside in a lower layer and thus the creation of these axioms preserves the stratifiability.

2. *Cycle-dependent axioms:* For every $v_i, v_j \in V$ with $i \neq j$ it needs to hold that v_i only occurs positively in the body of an axiom with head v_j, if such an axiom exists. The original set of axioms is already stratified, which means that this already holds. The axioms A^{unroll} do not change a positive occurrence of a derived variable to a negative one and thus preserve the stratifiability.

3. *Propagation axioms:* For every $v \in V$ we have a new axiom $v^{(k+1)} = T \leftarrow v^{(k)} = T$. This axiom always requires a positive occurrence of $v^{(k)}$ and thus preserves the stratifiability.

Thus, we have confirmed that none of the methods used to create unrolling axioms break the stratification, and therefore they can be added to the same layer as V_L. □

3.1.2 Fixed Point for Unrolling Axioms

We already showed that S_n is the fixed point for the original axioms A_V of SCC V, but only assumed that S_n^{unroll} is the fixed point for the unrolling axioms A_V^{unroll} without proof.

Theorem 2. *The fixed point for the unrolling variables A_V^{unroll} for SCC V is S_n^{unroll}.*

Proof. We define the sequence of all partial assignments for the unrolling variables as $S_0^{\text{unroll}}, S_1^{\text{unroll}}, \ldots$. Let S_0^{unroll} be a partial assignment for every variable $v^p \in vars(\varphi) \setminus \{V \cup V^{\text{unroll}}\}$ that occurs in axioms $(h = T \leftarrow \varphi) \in A_V^{\text{unroll}}$ and for every $v \in V \cup V^{\text{unroll}}$. The values for the variables v^p depend on what has been derived in previous axiom layers and are therefore fixed. The values for v are initialized as F. We further define S_1 as shown in Eq. (7), since we only evaluate the *cycle-independent axioms* in step 1. Since we set the values for variables with timestamp 0 in step 1, we have to shift all the indices by 1.

$$S_1^{\text{unroll}}(v) = \begin{cases} S_0^{\text{unroll}}(v) & \text{if } v \notin V \cup V^{\text{unroll}} \\ T & \text{if } (v^{(0)} = T \leftarrow \varphi) \in A_V^{\text{unroll}} \text{ with } S_0^{\text{unroll}} \models \varphi \\ F & \text{otherwise} \end{cases} \tag{7}$$

We define S_{k+1}^{unroll} for $k \in \{2, \ldots, n-1\}$ as shown in Eq. (8),

$$S_{k+1}^{\text{unroll}}(v) = \begin{cases} S_k^{\text{unroll}}(v) & \text{if } v \notin V \cup V^{\text{unroll}} \\ S_k^{\text{unroll}}(v) & \text{if } v := v^{(t)} \in V^{\text{unroll}} \text{ with } t \in \{0, \ldots, k-1\} \\ T & \text{if } (v^{(k)} = T \leftarrow \varphi^{(k-1)}) \in A_V^{\text{unroll}} \text{ with } S_k^{\text{unroll}} \models \varphi^{(k-1)} \\ F & \text{otherwise} \end{cases} \tag{8}$$

Since unrolling axioms are always of the form $v^{(k+1)} = T \leftarrow \varphi^{(k)}$, they only depend on the values that were derived in a previous timestamp or that are in previous layers. Therefore, if a timestamp k got evaluated, it will not change anymore and the evaluation will move on to the next timestamp $k+1$. The variables that belong to timestamps after $k+1$ will not yet be evaluated, and they are set to F.
The logic remains the same for the final step n, but the axioms $v^{(n-1)} = T \leftarrow \varphi^{(n-2)}$ set the values for the original variables $v \in V$ rather than the unrolling variables (this follows directly from our axiom definition in Section 3.1). Since we evaluate these variables, which only appear as the head of axioms in A_V^{unroll}, after n steps and all variables from earlier timestamps are fixed, we reach a fixed point where $S_n^{\text{unroll}} = S_{n+1}^{\text{unroll}}$.

□

3.1.3 Exact Representation of a Cycle

To ensure that unrolling can be used to transform cyclic dependencies into an acyclic representation, it needs to preserve the exact semantics of the original cycle, meaning that the fixed point needs to be the same.

Theorem 3. *Unrolling a cycle for an SCC V ensures that $S_n(v) = S_n^{unroll}(v)$ for all $v \in V$.*

Proof. We show that $S_n(v) = T \iff S_n^{\text{unroll}}(v) = T$ holds for both directions:
Case 1 ($S_n(v) = T \Rightarrow S_n^{unroll}(v) = T$): We show by induction that for all $k \in \{0, \ldots, n-2\}$ and $v \in V$ it holds that $S_k(v) = T \Rightarrow S_k^{\text{unroll}}(v^{(k-1)}) = T$.
Induction basis ($k = 1$): The definition of S_0 initializes all $S_0(v) = F$ and S_0^{unroll} initializes all $v^{\text{unroll}} \in V^{\text{unroll}}$ as $S_0^{\text{unroll}}(v^{\text{unroll}}) = F$. The only variables we set to true from S_0 and S_0^{unroll} are the *cycle-independent variables*.

For every axiom $(v = T \leftarrow \varphi) \in A_V$ we created an axiom $(v^{(0)} = T \leftarrow \varphi) \in A_V^{\text{unroll}}$ if $vars(\varphi) \cap V = \emptyset$ holds. We only create these axioms for $v^{(0)}$ and do not change the body. Since we have initialized S_0 and S_0^{unroll} in the same way, we know that if $S_0 \models \varphi$, then $S_0^{\text{unroll}} \models \varphi$ and finally $S_1^{\text{unroll}}(v^{(0)}) = T$.
Induction hypothesis: Assume $S_k(v) = T \Rightarrow S_k^{\text{unroll}}(v^{(k-1)}) = T$.
Inductive step ($k \to k+1$): We show that $S_{k+1}(v) = T \Rightarrow S_{k+1}^{\text{unroll}}(v^{(k)}) = T$.
We analyze the different ways we create the unrolling axioms separately:

1. *Propagation axioms:* For every $v \in V$ we have an axiom $v^{(k)} = T \leftarrow v^{(k-1)} = T$. From the definition of S_k we know that if $S_k(v) = T$, then $S_{k+1}(v) = T$. Using the induction hypothesis, this means that if $S_k(v) = T$, then $S_k^{\text{unroll}}(v^{(k-1)}) = T$ and finally $S_{k+1}^{\text{unroll}}(v^{(k)}) = T$.

2. *Cycle-dependent axioms:* For every axiom $(v = T \leftarrow \varphi) \in A_V$ we created an axiom $(v^{(k)} = T \leftarrow \varphi^{(k-1)}) \in A_V^{\text{unroll}}$. Using the induction hypothesis, we know that $S_{k+1}(v) = T$ if $S_k \models \varphi$, then $S_k^{\text{unroll}} \models \varphi^{(k-1)}$ and finally $S_{k+1}^{\text{unroll}}(v^{(k)}) = T$.

Thus, we conclude that $S_{k+1}(v) = T \Rightarrow S_{k+1}^{\text{unroll}}(v^{(k)}) = T$ for all $v \in V$.
We need another step, since we only proved that this holds until S_{n-1}. The step from S_{n-1} to S_n is analogous to the inductive step, but instead of $v^{(n-1)}$ we use v, and conclude that $S_n(v) = T \Rightarrow S_n^{\text{unroll}}(v) = T$ holds for all $v \in V$.

Case 2 ($S_n(v) = T \Leftarrow S_n^{unroll}(v) = T$): We show by induction that for all $k \in \{0, \ldots, n-2\}$ and $v \in V$ it holds that $S_k(v) = T \Leftarrow S_k^{\text{unroll}}(v^{(k-1)}) = T$.
Induction basis ($k = 1$): The definition of S_0 initializes all $S_0(v) = F$ and S_0^{unroll} initializes all $v^{\text{unroll}} \in V^{\text{unroll}}$ as $S_0^{\text{unroll}}(v^{\text{unroll}}) = F$. The only variables we set to true from S_0 and S_0^{unroll} are the *cycle-independent variables*.
For every axiom $(v = T \leftarrow \varphi) \in A_V$ we created an axiom $(v^{(0)} = T \leftarrow \varphi) \in A_V^{\text{unroll}}$ if $vars(\varphi) \cap V = \emptyset$ holds. We only create these axioms for $v^{(0)}$ and do not change the body. Since we have initialized S_0 and S_0^{unroll} in the same way, we know that if $S_0^{\text{unroll}} \models \varphi$, then $S_0 \models \varphi$ and finally $S_1(v) = T$.
Induction hypothesis: Assume $S_k(v) = T \Leftarrow S_k^{\text{unroll}}(v^{(k-1)}) = T$.
Inductive step ($k \to k+1$): We show that $S_{k+1}(v) = T \Leftarrow S_{k+1}^{\text{unroll}}(v^{(k)}) = T$.
We analyze the different ways we create the unrolling axioms separately:

1. *Propagation axioms:* For every $v^{(k)} \in V^{\text{unroll}}$ we have an axiom $v^{(k)} = T \leftarrow \varphi^{(k-1)}$ where $\varphi^{(k-1)} := (v^{(k-1)} = T)$. We know that if $S_k^{\text{unroll}} \models \varphi^{(k-1)}$, then $S_k^{\text{unroll}}(v^{(k-1)}) = T$ and therefore $S_{k+1}^{\text{unroll}}(v^{(k)}) = T$. Using the induction hypothesis, this means that if $S_k^{\text{unroll}}(v^{(k-1)}) = T$, then $S_k(v) = T$ and finally $S_{k+1}(v) = T$.

2. *Cycle-dependent axioms:* For every axiom $(v = T \leftarrow \varphi) \in A_V$ we created an axiom $(v^{(k)} = T \leftarrow \varphi^{(k-1)}) \in A_V^{\text{unroll}}$. Using the induction hypothesis, we know that $S_{k+1}^{\text{unroll}}(v^{(k)}) = T$ if $S_k^{\text{unroll}} \models \varphi^{(k-1)}$, then $S_k \models \varphi$ and finally $S_{k+1}(v) = T$.

Thus, we conclude that $S_{k+1}(v) = T \Leftarrow S_{k+1}^{\text{unroll}}(v^{(k)}) = T$ for all $v \in V$.
We need another step, since we only proved that this holds until S_{n-1}. The step from S_{n-1}

to S_n is analogous to the inductive step, but instead of $v^{(n-1)}$ we use v, and conclude that $S_n(v) = T \Leftarrow S_n^{\text{unroll}}(v) = T$ holds for all $v \in V$.

We showed that $S_n(v) = T \iff S_n^{\text{unroll}}(v) = T$ holds. Since $V \subseteq \mathcal{V}^d$, we know that every $v \in V$ can only be either T or F. This means that for every variable where $S_n(v) \neq T$ and $S_n^{\text{unroll}}(v) \neq T$, it holds that $S_n(v) = F$ and $S_n^{\text{unroll}}(v) = F$. From this we follow that $S_n(v) = S_n^{\text{unroll}}(v)$. □

3.2 Improvements

There are several ways on how we are able to improve unrolling, which will be discussed in this section. These improvements are only to reduce the amount of memory (and maybe also processing time) needed, everything else stays the same.

3.2.1 Remove Unused Axioms

Not every axiom that is created from unrolling will necessarily be used. In fact, there are some axioms which can never be reached based on the semantics of the cycle. Consider the axiom $roomReachable3^{(1)} = T \leftarrow roomReachable2^{(0)} = T \wedge doorOpenable2 = T$. At $t = 0$ only the body of *cycle-independent axioms* will be satisfied. The axiom $roomReachable2 = T \leftarrow roomReachable1 = T \wedge doorOpenable1 = T$ depends on derived variable *roomReachable1*, that is part of the loop, which means that an axiom with head $roomReachable2^{(0)}$ will not exist and therefore the body of the axiom $roomReachable3^{(1)} = T \leftarrow roomReachable2^{(0)} \wedge doorOpenable2$ will never be satisfied.

The rules for which axioms can safely be removed and why this works will be discussed in Section 3.2.2.1. This is done with a reachability analysis after the creation of the unrolling axioms. This computation creates an additional overhead, and we describe an on-line way to implement this in the following.

A property of axioms is that they only directly depend on the variables of their body, which in turn only depend on axioms that have those as their head. For unrolling, this means that an axiom at step $k + 1$ only depends on unrolling variables that are set from axioms at step k with $k \in \{0, \ldots, n - 3\}$ and external variables. The axioms at $t = 0$ only depend on external variables. Since we do not know the value of the external variables, we start by creating all *cycle-independent axioms* for an SCC V. Afterward, we iteratively create the *cycle-dependent axioms* and the *propagation axioms* for $t = 1$, where we know that for an axiom $h = T \leftarrow \varphi$ all $vars(\varphi)$ are the heads for axioms created in $t = 0$ or not part of the same SCC. This is repeated until $t = n - 1$.

This implementation works completely uninformed and only takes into account whether an axiom has been created or not, and not whether its body will actually be satisfied.

The benefit of this improvement is that axioms without any usage will not be created but the fixed point of V will not be changed. This way we can reduce the memory needed to store the axioms A_V^{unroll}. A possible issue is that the calculation overhead could still be too large, even if the implementation is done on-line during the creation of the unrolling axioms.

3.2.2 Remove Unused Variables

Following from the removal of unused axioms, there now exist unrolling variables which have no axiom associated to it, meaning that these variables have no usage anymore. Since those variables will never be used, they can safely be removed. The concept and implementation is the same as for removing the unused axioms and since that can be done on-line, this can be done on-line during the creation of the unrolling axioms as well. In fact, these implementations work very well together.
The rules for which variables can safely be removed will be discussed in Section 3.2.2.1.
The benefits and downsides are the same as for Section 3.2.1.

3.2.2.1 Determining Unreachable Axioms and Variables

The most straightforward way to determining which axioms and variables are reachable is by performing a reachability analysis. The analysis will be done after the unrolling for the respective cycle has been finished. There are special rules on how to create a graph for this problem and how to determine whether an axiom or variable is reachable based on that graph.
The goal is to create a directed graph $G = (V_R, E_R)$ from the SCC V with its axioms A_V and determine with a reachability analysis the two sets $R_a \subseteq A_V$ and $R_v \subseteq V$ that contain an overapproximation of the reachable axioms and variables (= the only axioms and variables that need to be created with unrolling). The overapproximation comes from the fact that we ignore the values that were assigned to variables in earlier layers. They could be taken into consideration as well, which could lead to an even better improvement, but is not taken a look at here. Let vertices $V_R = \mathcal{V} \cup A_V^{\text{unroll}}$ and edges $E_R = E_a \cup E_v$ with $E_a = \{(y, x) \mid \exists x := y = T \leftarrow \varphi, x \in A_V^{\text{unroll}}, y \in \mathcal{V}^d\}$ and $E_v = \{(x, y) : \exists y \in vars(\varphi)$ for $x := h = T \leftarrow \varphi, y \in \mathcal{V}^d, x \in A_V^{\text{unroll}}\}$. In words, this means that we have an edge from each axiom to its head variable and an edge from each derived variable to the axioms that contain this derived variable in any of its conditions.
To make this graph more readable when drawn, it can be split in a different layer for each timestamp, alternating variables and axioms, but it should start with the variables that were not created from the unrolling into the cycle-independent axioms and start with timestamp $t = 0$.
Now that the graph is constructed, a reachability analysis will be performed on it with the following steps:

1. Mark all variables $v = \mathcal{V} \setminus V^{\text{unroll}}$
2. Mark all arrows that are going out from the marked variables
3. Mark axioms where all incoming arrows are marked, add those to R_a
4. Mark all arrows that are going out from the marked axioms
5. Mark variables where at least one arrow is incoming, add those to R_v
6. Repeat from step 2 until the last timestamp is reached

We take a look at our example cycle $R_1 \to R_2 \to R_3 \to R_1$ in Figure 3.2 (we omit the lines from D_i after $k = 0$ since that would make the graph harder to read), the axioms are labeled

with their number from Eq. (6). The reachability analysis has already been performed and the variables and axioms that are marked in blue are reachable.

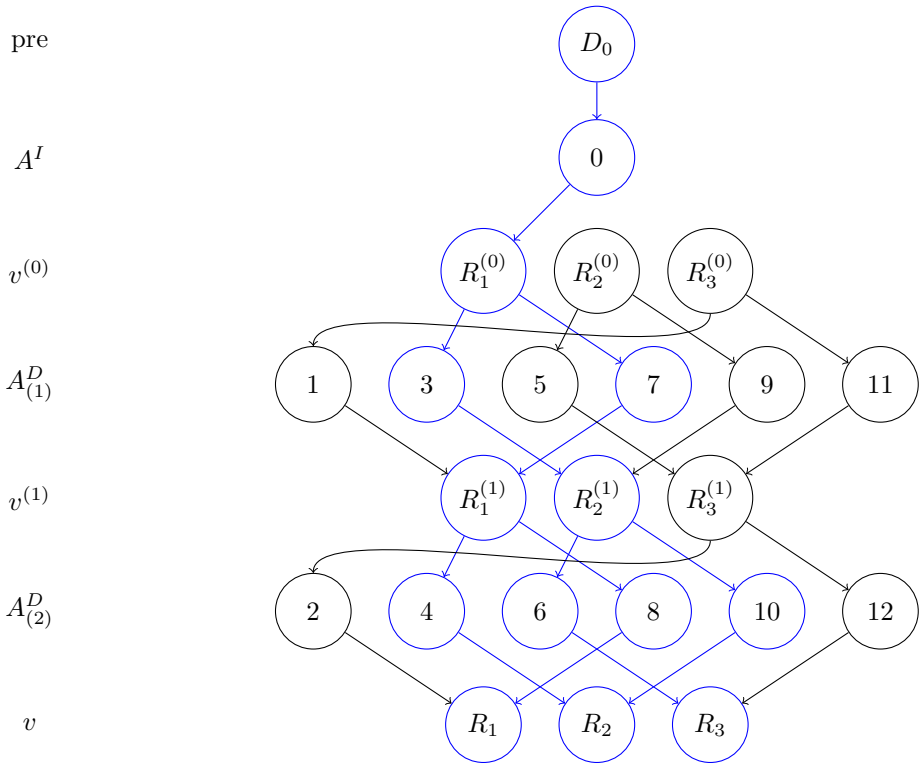


Figure 3.2: Dependency graph for axioms Eq. (6) showing reachable variables and axioms

Since this is done uninformed, all variables that were not created from unrolling are considered as reachable. An axiom can only be reached if all atoms of its body evaluate to true, else its body will never be satisfied, therefore making it unreachable. A derived variable § be reached if it is set to true by any axiom.

3.2.3 Replace Propagation Axioms with more Cycle-Independent Axioms

Propagation axioms are actually less important than one might think. Technically, their only purpose is to propagate the value of the cycle-independent axioms. Since we only create one copy for those axioms (e.g. $h^{(0)} = T \leftarrow \varphi$), they only set the value for $h^{(0)}$. The values of $h^{(k)}$ for $k \in \{1, \ldots, n-2\}$ are set by the propagation axioms.

What about the derived variables that are set by the cycle-dependent axioms then? They do not make use of the propagation axioms. We have that in the sequence of partial assignments created from unrolling $S_0, S_1, \ldots$ the set of variables with $S_k(v) = T$ is monotone, this means that if a variable $S_k(v) = T$ it also holds that $S_{k+1}(v) = T$. We take a look at Figure 3.1, remove the propagation axioms and add the missing cycle-independent axioms. The result is Figure 3.3.

The graph shows that we can still reach all variables R_1, R_2 and R_3 without the propagation axioms.

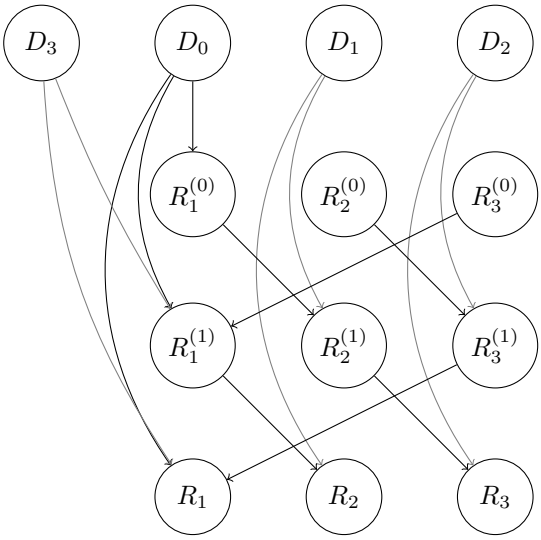


Figure 3.3: Dependency graph for axioms Eq. (6)
without propagation axioms but all cycle-independent axioms

There is also another way to look at this: Since we know that we only need to propagate the value of cycle-independent axioms, we could only create propagation axioms for those, i.e. only create axioms $v^{(k+1)} = T \leftarrow v^{(k)} = T$ for every $v \in V^I$ with $V^I = \{h \mid (h^{(0)} = T \leftarrow \varphi) \in A^I\}$.

3.3 Implementation

The implementation is done on top of Fast Downward 24.06 [Helmert, 2006].
Unrolling is implemented as a task transformation, which adds explicit axioms for how the default value of variables can be achieved. A task transformation transform the task in a way that the heuristic can use it, since for some heuristics the original axioms are not sufficient. The real implementation does not use false or true, but rather the default and non-default value, where the default value describes the original value of a variable. Since we add axioms that set a variable to its default value, we call them default value axioms. There are already two different methods implemented that create these default value axioms. We will discuss them in more detail in Chapter 4.
Unrolling is a third approach as to how to handle the creation of those default value axioms, which tries to exactly compute the cycles, instead of the approximation that has been done until now.
The implementation has been done as described in Section 3.1. It is implemented as a pre-calculation to transform the cyclic dependencies into an acyclic representation before the default value axioms are created. This way, the previous functionality of creating these axioms for derived variables that are not part of cycles can be used without having to worry about cyclic dependencies.

4

Related Work

We know that this chapter is usually used to talk about other publications and research that is related to our topic. But since there are no related papers that talk about related topics to unrolling cyclic dependencies for axioms, we will take a closer look at how achieving the default value for axioms has been handled until now. There are no published papers about these methods, instead they were directly implemented in Fast Downward [Helmert, 2006]. Something to note is that not all derived variables need an axiom that sets their default value. Before the creation of those, a dependency analysis is done, which tracks for all derived variables which of their values are needed. This analysis backtracks the default and non-default dependencies of a variable to check for which variables a default value axiom needs to be created. If there are cyclic dependencies for the non-default dependencies, this computation will never terminate and will loop forever. Therefore, we skip variables with cyclic dependencies in that computation with the approximations.
Since not every variable needs a default value axioms, we define the set $V^{\text{def}} \subseteq \mathcal{V}^d$ as the variables that need a default value and will use that for the following.
There are two methods for how those default value axioms are created, which are called "approximate negative" and "approximate negative cycles". We will first talk about approximate negative.
Approximate negative is the simplest way to handle the creation of default value axioms. We overapproximate the negative axioms for all variables that need their default value. This is done by creating new axioms $v = F \leftarrow \emptyset$ for every $v \in V^{\text{def}}$, which indicates that the default value for these variables can be achieved for free. This overapproximation can have a negative impact on the heuristic values since we essentially lose information about how the default value for these axioms is normally achieved.
Approximate negative cycles does the same overapproximation as approximate negative, but only for derived variables that have cyclic dependencies and where we need their default value. For all other derived variables, the negative axioms are computed exactly. For a derived variable v with n axioms $v = T \leftarrow \varphi_1, \ldots, v = T \leftarrow \varphi_n$ we add axioms that together represent $v = F \leftarrow \neg\varphi_1 \wedge \cdots \wedge \neg\varphi_n$. Since φ is a conjunction of atoms that e.g. looks like $\varphi := x_0 = T \wedge \cdots \wedge x_m = T$, $\neg\varphi$ would be a disjunction that e.g. looks like $\neg\varphi := \neg(x_0 = T) \vee \cdots \vee \neg(x_m = T) \equiv x_0 = F \vee \cdots \vee x_m = F$. Since we get

$v = F \leftarrow \neg\varphi_1 \wedge \cdots \wedge \neg\varphi_n$ and every $\neg\varphi$ is a disjunction, this creates a conjunction of disjunctions (CNF). The axioms in the planner are defined as disjunctions of conjunction (DNF), therefore we essentially need to convert a CNF to a DNF, which can cause this approximation to be slow. But since we compute these default value axioms exactly, we do not lose any information like we did for the approximate negative method.

5
Experimental Analysis

In this chapter we will analyze the performance of the unrolling implementation compared to the original implementation. There are two possible options "approximate negative" and "approximate negative cycles" in the original implementation, which we will both compare to our unrolling implementation. The search algorithm we used for our experiments is eager greedy search with the heuristics h^{add} [Bonet and Geffner, 2001] and h^{FF} [Hoffmann and Nebel, 2001]. Additionally, we used the lazy greedy [Richter and Helmert, 2009] search with the landmark count heuristic h^{LM} [Richter and Westphal, 2008] with landmarks extracted using the RHW method [Richter et al., 2008] and ordered using the reasonable ordering [Hoffmann et al., 2004]. The experiments are run on the sciCORE cluster of the University of Basel with a time limit of 30 minutes and a memory limit of 3584MB. They are run on our implementation of unrolling, which is done on top of Fast Downward 24.06 [Helmert, 2006], using Downward Lab [Seipp et al., 2017], which is a Python package specifically created to run experiments with Fast Downward.
Since approximate negative and approximate negative cycles show the same trends when compared to unrolling, we will focus mainly on the comparison between approximate negative cycles and unrolling. The approximate negative method is usually worse than approximate negative cycles. But if there is anything interesting, we will point that out. We will also mainly focus on the results of the h^{add} heuristic. The additional plots are in Appendix A.

5.1 Benchmark Suites

We use specific benchmark suites to run these tests. They need to contain cyclic dependencies since unrolling only triggers for cycles. Therefore, we chose a set of four benchmark suites that all contain cycles and therefore will use unrolling: *psr-large*[1] [Hoffmann and Edelkamp, 2005], *psr-middle*[1] [Hoffmann and Edelkamp, 2005], *drones-horndl*[2] [Borgwardt et al., 2022] and queens-horndl[2] [Borgwardt et al., 2022].

[1] https://github.com/aibasel/downward-benchmarks
[2] https://zenodo.org/records/ 12624112

We roughly classify these suites based on what kind of cycles they have:

- *drones-horndl:* Only has cycles of size $|V| = 2$.
- *queens-horndl:* Has a lot of cycles of size $|V| = 2$ and also cycles of size $|V| > 100$.
- *psr-middle:* Only has few cycles, where the size varies, but they are usually of size $|V| < 60$.
- *psr-large:* Similar to *psr-middle* but there are some problems with cycles of size $|V| > 300$.

5.2 New Axioms

We will take a look at how many new axioms will roughly be created with unrolling. We approximate the expected number of new axioms from SCC V with axioms A_V as seen in Eq. (9).

$$\begin{aligned} |A^{\text{unroll}}| &= |A^P| + |A^D| + |A^I| \\ &= |V| \cdot (|V| - 1) + |A_V| \cdot (|V| - 1) + |A^I| \\ &\approx |V|^2 + |A_V| \cdot |V| + n_I \overset{|A| \geq |V|}{\geq} 2 \cdot |V|^2 + |A^I| \end{aligned} \tag{9}$$

The number of cycle-independent axioms we create is a constant, which cannot be approximated with a formula. This means, that the number of axioms increase by a factor of $|V|$ squared. The consequences can be seen in Table 5.1.

Benchmark	A^{def} (neg)	A^{def} (neg cyc)	A^{def} (unroll)	A^{unroll}
drones-horndl	1%	66%	63%	5%
queens-horndl	0%	4%	39%	60%
psr-middle	0%	9%	4%	80%
psr-large	0%	5%	2%	80%

Table 5.1: Percentages of default value axioms (A^{def}) and unrolling axioms (A^{unroll}) compared to all axioms
(neg = approximate negative, neg cyc = approximate negative cycles & unroll = unrolling)

We need to differentiate between how many unrolling axioms A^{unroll} and default value axioms A^{def} are created. These are directly connected to each other, since if we have more axioms from unrolling, we have more variables and that means that there may be more variables where the default value is needed.
Table 5.1 shows that only for *drones-horndl* the unrolling axiom percentage is lower than the default value axioms percentage and even before unrolling, the default value axioms percentage is really high. *Queens-horndl* is mainly dominated by the unrolling axioms, but also has a lot of default value axioms. Finally, *psr-middle* and *psr-large* are roughly the same, where the unrolling axioms dominate and there are not a lot of default value axioms in comparison.

Something to note is that a higher default value axioms percentage for the approximations either means that a lower percentage of the total axioms has cyclic dependencies or that the default value for fewer of those is needed. This happens because the implementation for the approximations filters out some variables in cycles at the very start and does not even consider whether their default value is needed.

To further illustrate the computational consequences, we take a look at an example *queens-horndl* problem with 10,000 initial axioms. Since we know that these only make up 1% of all the axioms we have after unrolling, we know that we will create approximately 1,000,000 new axioms with unrolling. These are roughly split into 600,000 unrolling axioms and 400,000 default value axioms. They both use the same data structure, and we assume that they each need about 100 bytes of storage. For every two unrolling axioms we create on average one new unrolling variable (every unrolling variable is usually the head of one propagation axiom and one cycle-dependent axiom). We assume that they need about 50 bytes each. This means, that we created a total of $1,000,000 \cdot 100\text{B} + 300,000 \cdot 50\text{B} = 115\text{MB}$ memory for this problem. While 115MB does not sound like a lot, if we compare it to the original 1MB needed to store the initial axioms it represents an increase of two orders of magnitude.

5.3 Initial h Value

The initial h value is the heuristic estimate for the initial state and depends directly on the used heuristic. If the initial heuristic value is 0, the search essentially starts off as a blind search since the heuristic is uninformed at the start.

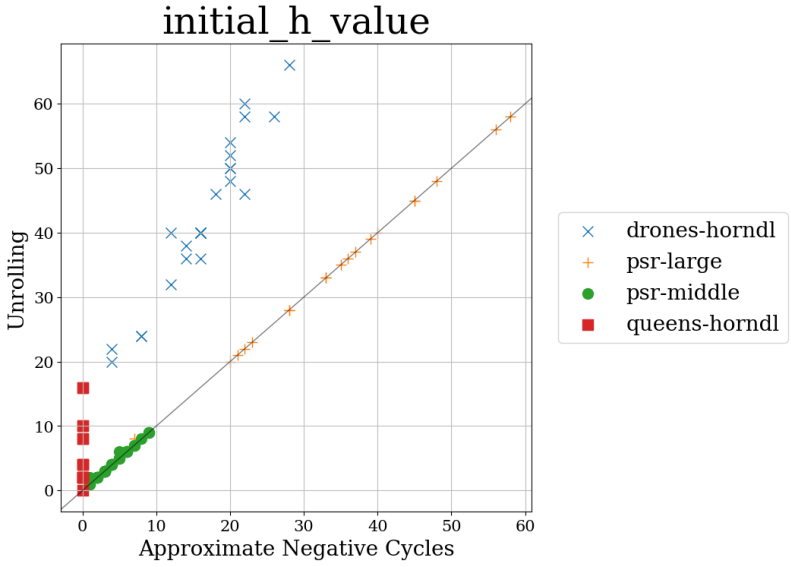


Figure 5.1: Scatter plot that compares the initial h value for unrolling and approximate negative cycles for h^{add}

Figure 5.1 shows the comparison of the initial heuristic values of unrolling and the approximation. The figure shows that the values are always higher for unrolling than for the

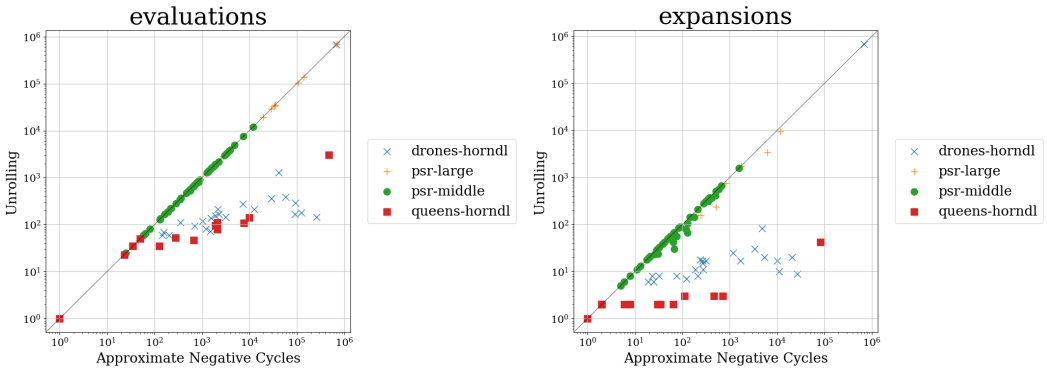
approximation. This is mainly seen for *drones-horndl* and *queens-horndl*, while *psr-middle* and *psr-large* show barely any change. The change of the initial heuristic value is not unusual and was definitely to be expected, since the approximation assigns the cycle variables the default value for free. This loss of information is reflected in a lower heuristic value. Unrolling on the other hand computes the default values of the cycle variables exactly, which preserves the information. Since the heuristics for unrolling have more information, the heuristic values are a closer representation of the actual costs. But there are also a lot of domains, where the initial heuristic value does not change at all. Which means that the information that is preserved from unrolling not necessarily has to change the values of the heuristic.

Both h^{FF} and h^{LM} show the same change, but the difference is definitely the biggest for h^{add}. There are some rare cases where the initial value is actually lower for unrolling for these heuristics.

Something to note is that there are some domains that have an initial heuristic value of 0 with the approximation. Especially for the approximate negative method every domain for h^{add} and h^{FF} has an initial heuristic value of 0. This stems most likely from the fact that it assigns every derived value its default value for free.

5.4 Evaluated and Expanded States

The generated states are all states that are generated during the search. The evaluated states are a subset of the generated states and are the states whose heuristic value was calculated to find the most promising candidate. The most promising candidate is the state that gets further expanded, and its successor states are generated. Since the generated and evaluated states are the same for an eager greedy search, we will not take a look at the generated states. In Section 5.3 we saw that the initial heuristic values changed, which means that the search could be different and therefore the number of state evaluations and expansions might change.



(a) evaluated states h^{add} (b) expanded states h^{FF}

Figure 5.2: Scatter plots that compare the evaluated and generated states for unrolling and approximate negative cycles for h^{add}

Figure 5.2 shows the comparison of the state evaluations and expansions of unrolling and the approximation. The figures show that in general fewer states are expanded and evaluated. There are many cases where the values are the same, which means that unrolling did not have an effect on those domains. As for the initial heuristic value, there are big differences for most *drones-horndl* and *queens-horndl* problems, while *psr-middle* and *psr-large* show barely any difference. There are some cases where the value is higher for both h^{FF} and h^{LM}, which means that unrolling can have a negative effect on those heuristics. This only happens in very specific cases though, since in general unrolling performs better for these heuristics as well.
The results directly depend on our observation in Section 5.3. Unrolling makes the search more informed and therefore needs fewer expansions to expand a goal state. The search stops at the first expanded goal state since we are not using an optimal search. Thus, we need to evaluate fewer states in total to find a solution. For domains where the initial heuristic value does not change, the search stays the same, which means that also the same number of states are evaluated and expanded.

5.5 Memory

Memory describes how much storage is needed for the search. This includes what is stored during an experiment, which are the states that are evaluated and expanded during the search, the axioms and the variables. These axioms include the original axioms, but also the ones created from the task transformations, which are the default value axioms and the unrolling axioms in our case. The variables include the initial variables and those created from unrolling. We discussed how many new axioms and variables we create in Section 5.2.

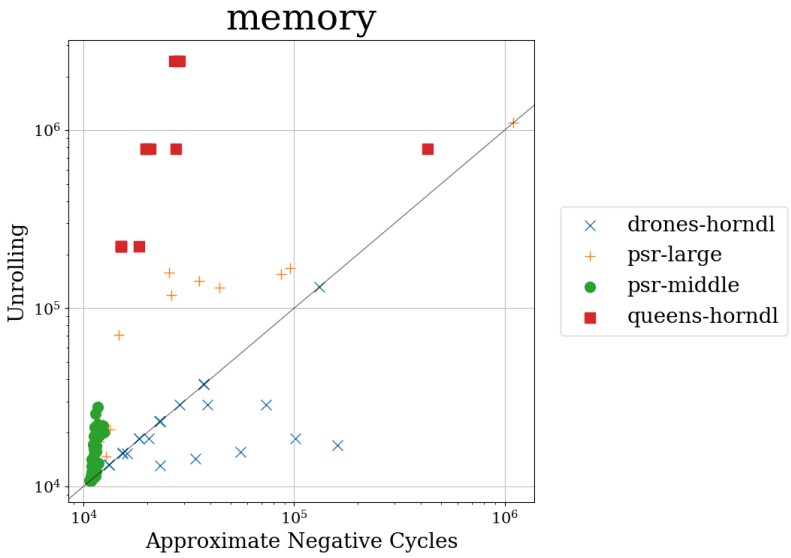


Figure 5.3: Scatter plot that compares the memory needed for unrolling and approximate negative cycles for h^{add}

Figure 5.3 shows the comparison of the memory needed for unrolling and the approximation. The figure shows that in general a lot more memory is needed for unrolling. Only for *drones-horndl* less memory is needed on average. The results are the same for both approximation methods and for all heuristics and directly reflect what we discussed in Section 5.2. The results for approximate negative cycles with h^{LM} are slightly different and show that not even *drones-horndl* needs less memory compared to that configuration.
The amount of memory needed is a combination between the states, axioms and variables that are stored during an experiment. Section 5.4 showed that we need to generate fewer states in general. This reduces the memory needed to store the states. But this reduction is a lot less than what the unrolling axioms and variables we create need in turn. Only for *drones-horndl*, where we do not create that many new axioms, the result is that less memory is needed, but only for some of the domains.

5.6 Total Time

The total time describes how long it takes to initialize and run the search until a goal state is found. This includes the task transformations, the heuristic values that are calculated for the search and the search itself. While the task transformations increase the total time and the heuristic calculations may take longer since we have more axioms, the time saved from fewer state expansions may balance the total time needed out.

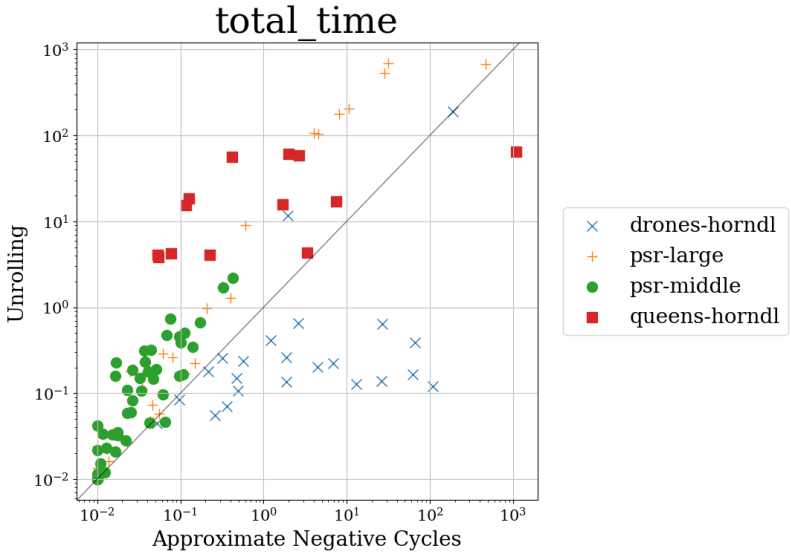


Figure 5.4: Scatter plot that compares the total time needed for unrolling and approximate negative cycles for h^{add}

Figure 5.4 shows the comparison of the total time needed for unrolling and the approximation. The figure shows that in general more time is needed for unrolling. Once again, only *drones-horndl* needs less time on average and there are a few *psr-middle* and *queens-horndl* problems that need less time as well.
Section 5.4 showed that we need fewer state expansions, but the total time needed for un-

rolling is still more for most of the domains. This means that the task transformation takes up more time than the search can save with the new axioms. The results for approximate negative cycles with h^{LM} are slightly different and show that not even *drones-horndl* needs less time compared to that configuration.

5.7 Cost

The cost describes how expensive a solution is. The heuristics we use are not admissible and eager greedy search never guarantees optimal solutions. The initial heuristic values are different, and some domains need fewer state expansions, which means that there could possibly be different solutions that are found with unrolling.

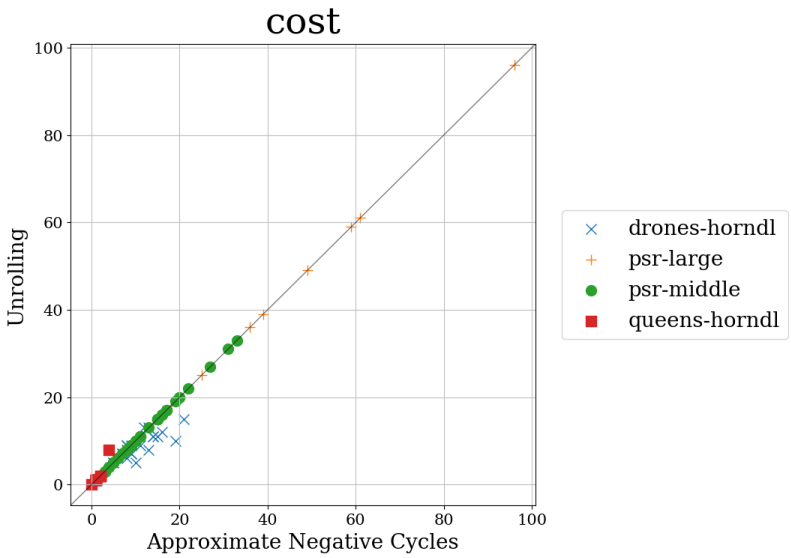


Figure 5.5: Scatter plot that compares the solution cost for unrolling and approximate negative cycles for h^{add}

Figure 5.5 shows the comparison of the cost of the found solution for unrolling and the approximation. The figure shows that in general the solution costs stay the same. This is especially true for *psr-middle* and *psr-large*, since those barely show any change. *Drones-horndl* has some variation between the different configurations. For approximate negative cycles with h^{add} and h^{FF} the solution cost is reduced on average, while for all other configurations the solution cost is on average higher.

The main reason as to why the solution costs are higher than for approximate negative could be the initial heuristic value. Since that value is 0 for all domains, it starts as a blind search. If the heuristic values stay 0 for future state evaluations, that would be an entire blind search, which results in optimal solutions. Since we do not know the heuristic values apart from the initial one, this is only an assumption.

The difference in solution costs compared to the ones from approximate negative cycles most likely stems from the different heuristic values during the search, which result in finding a solution with fewer expansions. That solution can have a lower cost but also a higher one,

which depends solely on how the heuristics interact with the new axioms.

5.8 Coverage

The coverage describes for how many problems a solution has been found. We will analyze each heuristic separately.

Benchmark	Approximate Negative h^{add}	Approximate Negative Cycles h^{add}	Unrolling h^{add}
drones-horndl	22	22	23
queens-horndl	36	36	32
psr-middle	50	50	50
psr-large	17	17	15
total	125	125	120

Table 5.2: Coverage for h^{add}

Table 5.2 shows the coverage for the different domains and methods used for the heuristic h^{add}. The coverage for approximate negative and approximate negative cycles is the same, while the coverage for unrolling varies by domain. For *drones-horndl* the coverage is better but for the *psr-large* and *queens-horndl* the coverage is worse.

Benchmark	Approximate Negative h^{FF}	Approximate Negative Cycles h^{FF}	Unrolling h^{FF}
drones-horndl	22	23	23
queens-horndl	17	17	16
psr-middle	44	44	43
psr-large	17	17	15
total	100	101	97

Table 5.3: Coverage for h^{FF}

Table 5.3 shows the coverage for the different domains and methods used for the heuristic h^{FF}. The coverage for approximate negative and approximate negative cycles is almost the same, while the coverage for unrolling varies by domain. For every domain apart from *drones-horndl* the coverage is worse than for the approximations.

Benchmark	Approximate Negative h^{LM}	Approximate Negative Cycles h^{LM}	Unrolling h^{LM}
drones-horndl	23	23	23
queens-horndl	41	41	32
psr-middle	50	50	50
psr-large	23	30	11
total	137	144	116

Table 5.4: Coverage for h^{LM}

Table 5.4 shows the coverage for the different domains and methods used for the heuristic h^{LM}. The coverage for approximate negative is worse than the coverage for approximate negative cycles. The coverage for unrolling is a lot worse than the approximations, but only for *psr-large* and *queens-horndl*.

These results show exactly what we had discussed before. Since we set a limit for both memory and time in our experiments, the problems where no solution was found ran either out of time or memory. This directly follows our observations of Section 5.5 and Section 5.6 respectively. Since both *psr-large* and *queens-horndl* have big cycles, we create a lot of unrolling axioms, which take both a lot of memory to store and time to create. Since that overhead exceeds our limits, the search terminates with no solution.
The domains with smaller cycles did not exceed the memory and time limits and therefore do not show much change in the coverage. *Drones-horndl* even has a better coverage for h^{add}.
Something to note is that the coverage for h^{LM} is a lot worse with unrolling, which means unrolling should not be used in combination with the landmark heuristic. The approximate negative cycles method already produces good results with this heuristic.

5.9 Discussion

In this section, we will take a closer look at how unrolling performed in general and what trade-offs were made.

The first factor we will discuss is how the cycle size impacts the results. This is the most significant factor, which influences the performance of unrolling the most. We make a distinction between small and big cycles.
The best example for a domain with small cycles is *drones-horndl*. All cycles were of size $|V| = 2$, which means that there were not many axioms that are created from unrolling. The quadratic growth of the axioms is negligible here. Even though there were not many new axioms created, they held essential information for the heuristics.
In contrast, *queens-horndl* had a lot of big cycles, where some were of size $|V| > 100$. While the heuristics were able to profit from these new axioms, there was a massive memory and preprocessing time overhead. The quadratic growth of axioms had a visible effect, making unrolling less effective despite the overall more accurate heuristic. Since that domain also

had a lot of cycles of size $|V| = 2$, they may have actually had a higher impact on the more accurate heuristic than the big cycles.
This observation suggests that the cycle size is very important and small cycles might have a bigger influence than big cycles. This could lead to a hybrid implementation, where only small cycles are computed exactly, and big cycles use the approximation.

The second factor we will discuss is how the information we preserved with unrolling is reflected in the results. The results from *psr-middle* and *psr-large* illustrate cases where the cost of unrolling shows no improvements at all. The total time and memory that is needed is increased significantly with barely any benefits. This suggests that the information gained from unrolling had no active effect on the heuristics and therefore perform roughly the same with the approximations. In such cases, unrolling only adds a computational overhead, which results in an overall decrease of performance.
This observation shows that not all domains will necessarily profit from unrolling. Therefore, it is important to know the domains before deciding on whether it is worth to use unrolling or not.

Finally, we need to take a look at what we were able to observe from Table 5.1. There is an interesting correlation between *drones-horndl* and *queens-horndl*, which are the suites that profited from unrolling the most. Both have a rather high percentage of default value axioms, which means that they have a lot of variables whose default value is important. This indicates that the new unrolling axioms were actively used to derive the necessary default values. On the other hand, both PSR suites have a low percentage of default value axioms, which in turn means that they did not make use of the new unrolling axioms to derive necessary default values.
This suggests a potential improvement, which is to only unroll cycles that have variables where the default value will be used from the heuristic. If the default value for all cycle variables is not relevant, there is no need to add the additional computation overhead from unrolling that cycle, and the approximation methods will be sufficient.

6 Conclusion

In this thesis, we addressed the challenge of handling cyclic dependencies between derived variables for delete relaxation heuristics. We successfully implemented unrolling as a method to transform these cyclic dependencies into an exact acyclic representation. We established proofs to show that the correct fixed point is reached while preserving stratifiability.
In our experimental analysis we compared our implementation of unrolling to the already existing approximation methods. The results show that unrolling successfully preserves the information of the cycles that is lost in the approximation, which resulted in a higher initial heuristic value and fewer state expansions for some domains. However, we also identified a possible overhead, which is the quadratic growth of the axioms compared to the size of the SCC it unrolls. To create and store these axioms, a lot of memory and processing time is needed, which can affect the domains negatively. As a result, unrolling is highly effective for domains with small cycles (e.g. *drones-horndl*), but can lead to an increase in both memory and preprocessing time for domains with big cycles (e.g. *queens-horndl*).

6.1 Future Work

Based on our findings, there are several opportunities for future research.

There are some improvements we described in Section 3.2 that could be tested. The main idea of those is to reduce the amount of memory needed by decreasing the number of unrolling axioms and unrolling variables that are created. Following the reduce of unrolling variables, there will also be fewer default value axioms that need to be created. Depending on how many axioms can be removed, the memory overhead might become smaller, making unrolling more effective for domains like *queens-horndl*. To further improve on our idea of Section 3.2.2.1, the analysis could be performed while considering all assignments to variables in previous layers. This way, even more axioms could be possibly removed, but the processing time might increase instead.

Apart from the improvements we previously discussed, the analysis showed even more parts that could be improved. Since the analysis has shown that unrolling is mostly effective when

the new axioms are used for the creation of new default value axioms, we could focus on only unrolling an SCC if its derived variables have default values that are used for the heuristic. Another improvement could be to only unroll small cycles, since the analysis showed that they create a negligible number of unrolling axioms, while big cycles have a huge effect on memory and preprocessing time. This would create a hybrid approach, where small cycles are unrolled and big cycles are approximated. The threshold for that distinction could be experimentally determined.

A last improvement could be to not unroll the full $n = |V|$ steps for every SCC, but only unroll for as many timestamps that are needed to reach the same fixed point as for the original cycle. This could be done by computing the longest simple path in the SCC and use this as n. Since the set of variables we set to true is monotone, a derived variable can change its value only once. Therefore, repeating a vertex in the path provides no new information, which means that the necessary depth is bounded by the longest path that does not repeat vertices.

Bibliography

Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1): 5–33, 2001.

Stefan Borgwardt, Jörg Hoffmann, Alisa Kovtunova, Markus Krötzsch, Bernhard Nebel, and Marcel Steinmetz. Expressivity of planning with horn description logic ontologies. In *Proc. AAAI 2022*, pages 5503–5511, 2022.

Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th International Planning Competition. Technical Report 195, University of Freiburg, 2004.

Claudia Grundke and Gabriele Röger. Eliminating negative occurrences of derived predicates from PDDL axioms. *arXiv*, 2510.14412, 2025.

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

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

Jörg Hoffmann and Stefan Edelkamp. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research*, 24:519–579, 2005.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS 2009*, pages 273–280, 2009.

Silvia Richter and Matthias Westphal. The LAMA planner — Using landmark counting in heuristic search. IPC 2008 short papers, http://ipc.informatik.uni-freiburg.de/Planners, 2008.

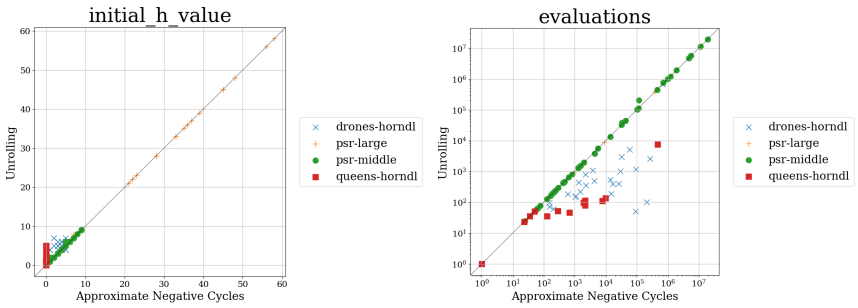
Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *Proc. AAAI 2008*, pages 975–982, 2008.

Gabriele Röger and Claudia Grundke. Negated occurrences of predicates in PDDL axiom bodies. In *Proceedings of the KI 2024 Workshop on Planning, Scheduling and Design (PuK 2024)*, 2024.

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

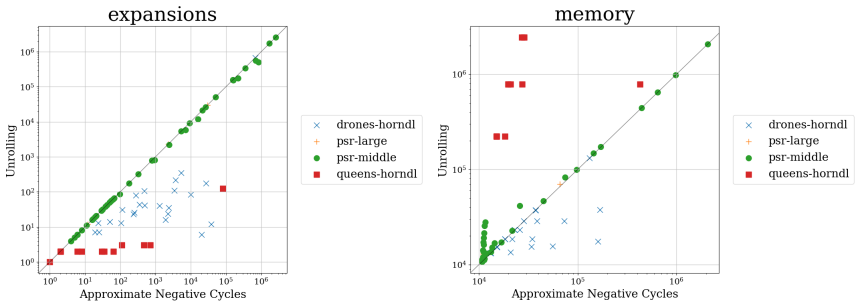
A

Additional Scatter Plots



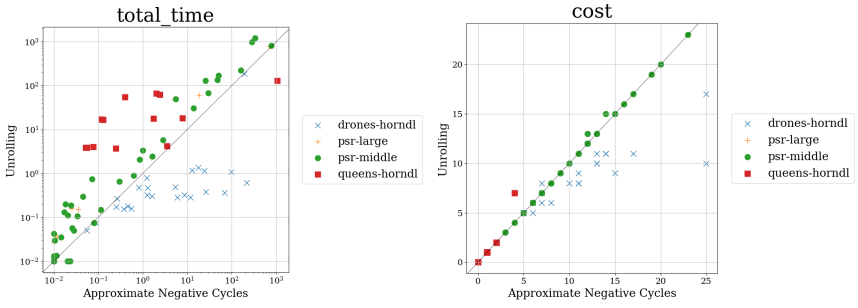
(a) initial h value h^{FF}

(b) evaluated states h^{FF}



(c) expanded states h^{FF}

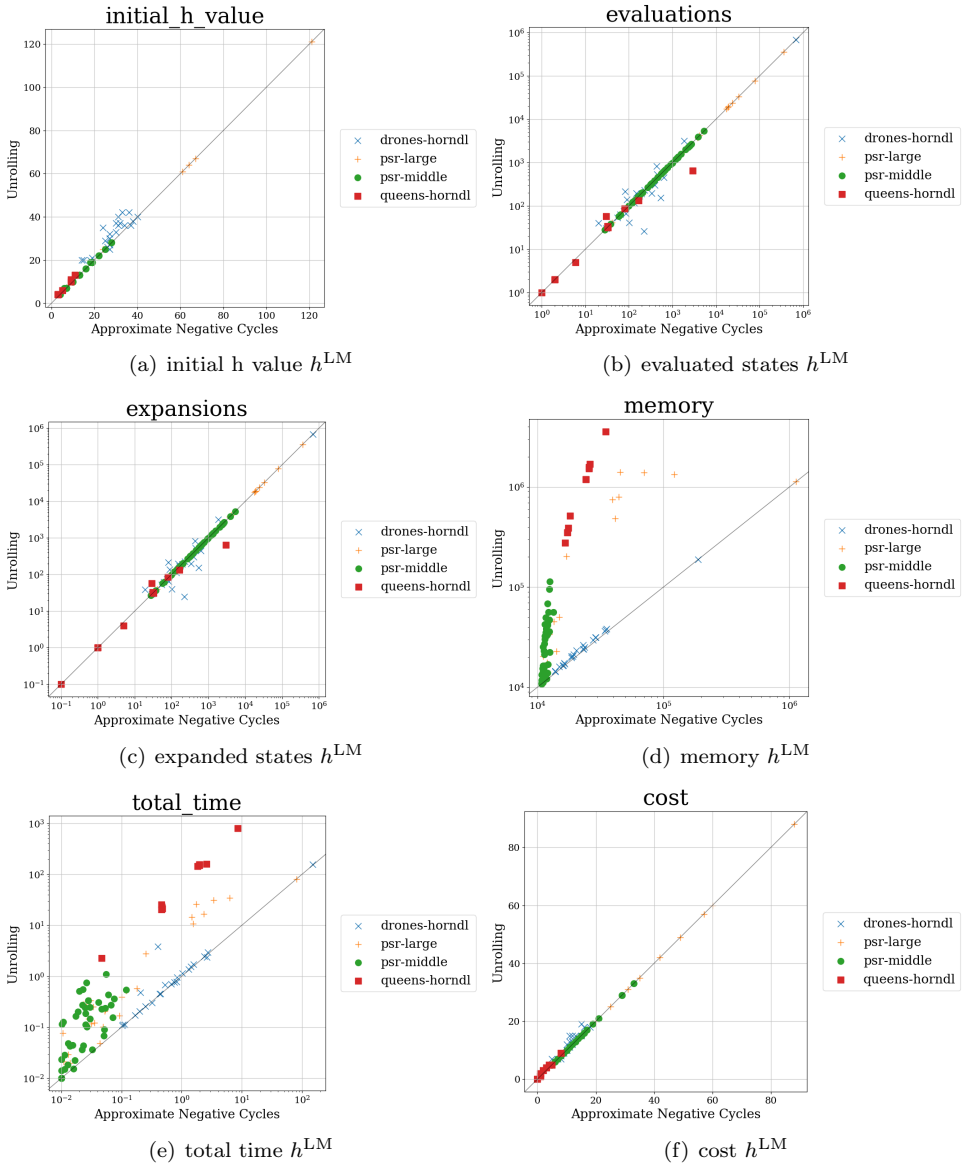
(d) memory h^{FF}



(e) total time h^{FF}

(f) cost h^{FF}

Figure A.1: Scatter plots that compare approximate negative cycles and unrolling for h^{FF}



(a) initial h value h^{LM}

(b) evaluated states h^{LM}

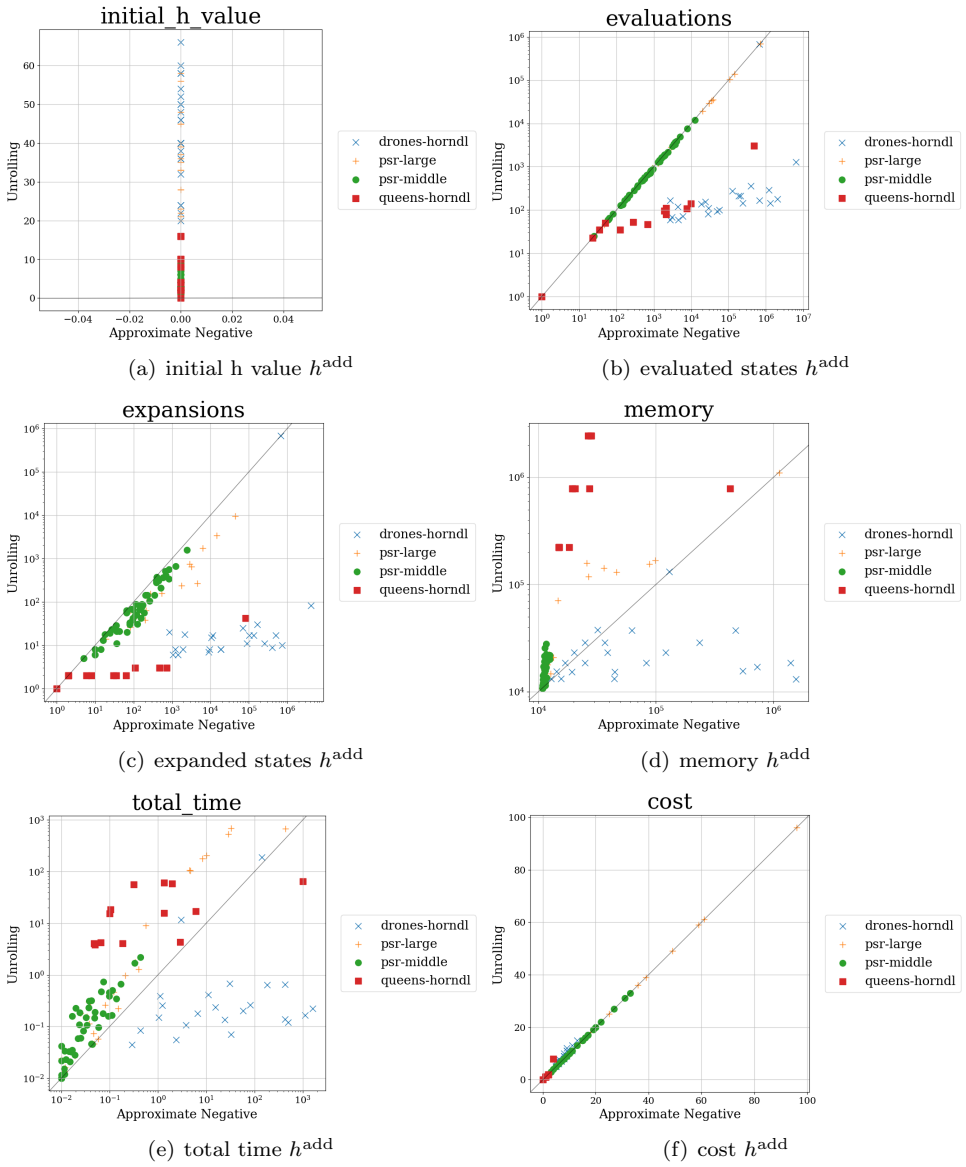
(c) expanded states h^{LM}

(d) memory h^{LM}

(e) total time h^{LM}

(f) cost h^{LM}

Figure A.2: Scatter plots that compare approximate negative cycles and unrolling for h^{LM}



(a) initial h value h^{add}

(b) evaluated states h^{add}

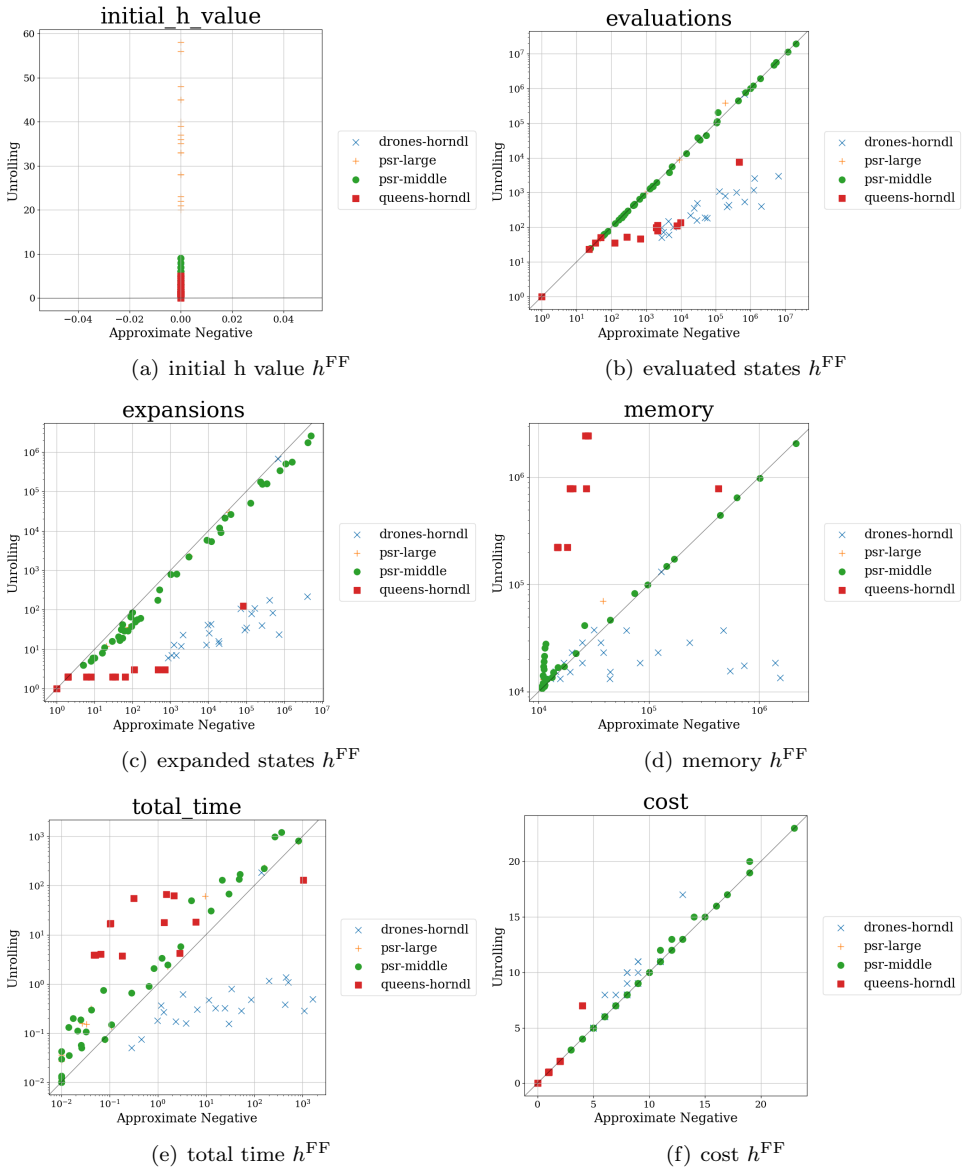
(c) expanded states h^{add}

(d) memory h^{add}

(e) total time h^{add}

(f) cost h^{add}

Figure A.3: Scatter plots that compare approximate negative and unrolling for h^{add}



(a) initial h value h^{FF}

(b) evaluated states h^{FF}

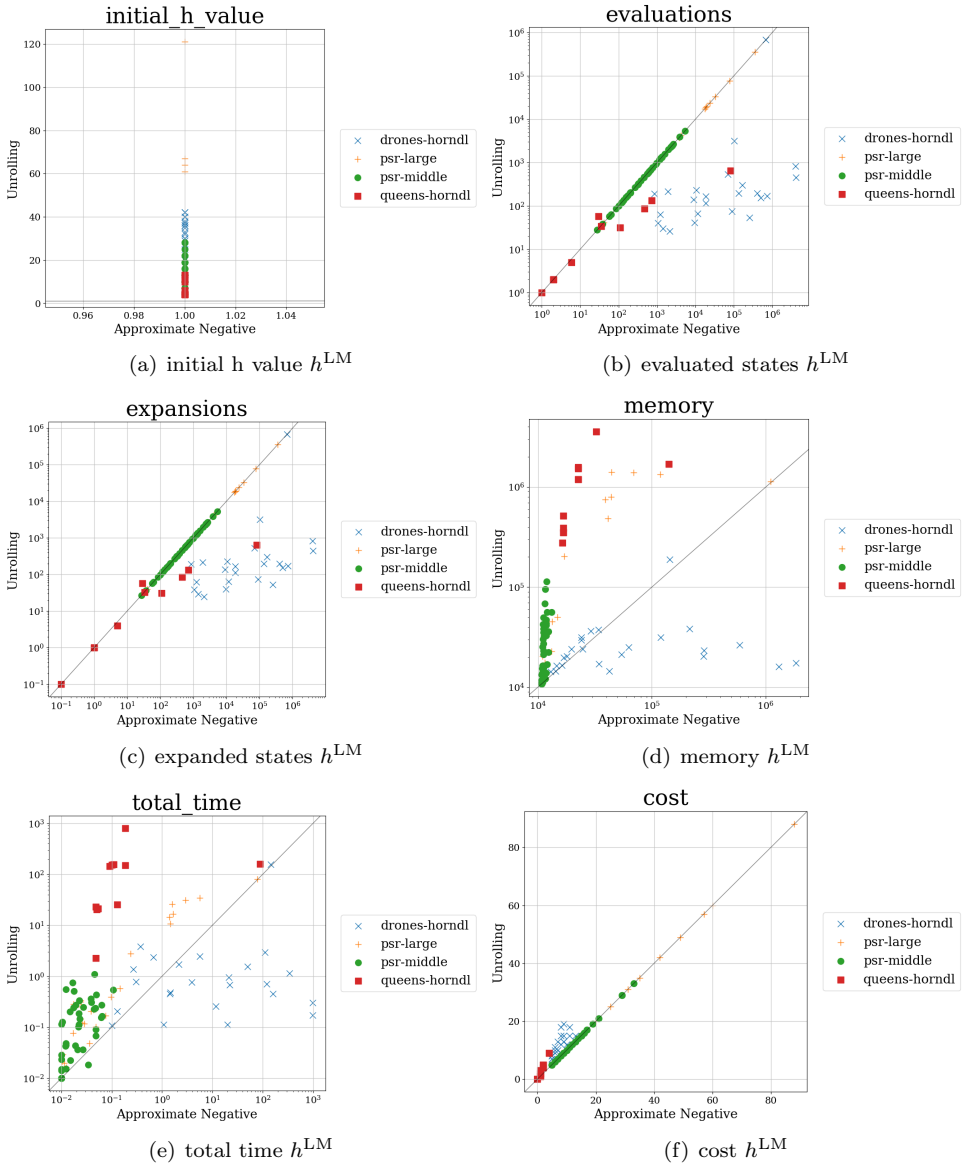
(c) expanded states h^{FF}

(d) memory h^{FF}

(e) total time h^{FF}

(f) cost h^{FF}

Figure A.4: Scatter plots that compare approximate negative and unrolling for h^{FF}



(a) initial h value h^{LM}

(b) evaluated states h^{LM}

(c) expanded states h^{LM}

(d) memory h^{LM}

(e) total time h^{LM}

(f) cost h^{LM}

Figure A.5: Scatter plots that compare approximate negative and unrolling for h^{LM}