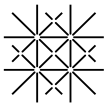


University of Basel

Optimizations for the Additive Heuristic in Fast Downward

Bachelor thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Salomé Eriksson

Simona Wittner
simona.wittner@stud.unibas.ch
2020-051-876

24.06.2024

Acknowledgments

I would like to express my gratitude to my supervisor Dr. Salomé Eriksson for the invaluable assistance, guidance, support and feedback throughout my thesis. I would also like to thank Prof. Dr. Malte Helmert for giving me the opportunity to do my thesis in the Artificial Intelligence Research Group and to work on an interesting topic.

Abstract

In classical planning, actions are used to reach a goal from the initial state. Search algorithms explore the state space of a planning problem to find a plan, which is a sequence of actions. To estimate the distance to the goal for the states, search algorithms can use heuristics. One heuristic used to guide the search is the additive heuristic. Recently, optimizations were presented to efficiently compute the additive heuristic in the context of lifted planning. Fast Downward is a classical planning system that uses a ground representation. In this work, the optimizations have been used and integrated into Fast Downward to see if they can be adapted to this ground planning system and whether they bring benefits. The optimizations are used here to reduce the number of unary operators required to calculate the heuristic, thereby enhancing performance. It was found that the optimizations can be used to compute the additive heuristic more efficiently for specific domains.

Table of Contents

1 Introduction

Classical planning, a subfield of artificial intelligence, is about identifying a sequence of actions to reach a specific goal from a specific initial state. All constraints or conditions defined by the problem must be satisfied. To identify such a sequence of actions, search algorithms are employed to explore the state space of a planning problem and examine sequences of actions and the resulting states. Search algorithms can use heuristics to enhance efficiency. Heuristics guide the search process in a targeted manner. They provide estimates of the cost or distance to reach the goal state from a state. These estimates assist the search in prioritizing, thereby minimizing the exploration of paths that are unlikely to lead to a satisfactory solution. This reduces the number of states to be explored, which is crucial for an efficient search. Certain heuristics are computed from solving a relaxed version of the original problem that ignores certain constraints. This helps to solve the problem more easily. The costs of the solutions to these simpler, relaxed problems are used for the estimates of the heuristic for the original problem. The heuristic considered in this thesis is the additive heuristic. It is a delete relaxation heuristic, which simplifies the original problem by ignoring the delete effects of actions. Once a fact has become true, it remains true. The additive heuristic estimates the distance to the goal in a search algorithm by adding up the cost of the subproblems, treating them as independent.

Planning problems are described and transformed into specific representations, which significantly impact how they are handled. The lifted representation uses variables that can later take on the values of objects. It is an abstract and therefore flexible and compact representation because the variables are instantiated if required. However, the use of a lifted representation is complex and requires specialized techniques and algorithms. With a ground representation, all abstract variables are replaced by concrete values or objects. This simplifies the calculation of state transitions and actions, as no further instantiations are required. However, if all possible combinations of variables and their values are explicitly represented, the resulting representation can become quite extensive. This is very memory-intensive, as the number of possible states and actions grows with the number of variables and objects, and it is very time-consuming due to the processing of all possible combinations. However, by analyzing the planning problem, it is possi-

ble to reduce the size of the representation. Helmert (2009) uses relaxed reachability analysis before grounding, so that only relevant objects and actions are considered. Logical programs, Datalog programs consisting of logical facts and rules, are used in this context.

Recently, an approach was introduced to compute certain heuristics from the lifted representation of the planning problem (Corrêa et al., 2021b). This approach uses extended Datalog programs with optimizations for the construction of the rules to compute the additive heuristic. In this thesis, this approach is adapted for the classical planning system Fast Downward (Helmert, 2006), which works on a ground representation. The additive heuristic is computed using unary operators, which are ground actions with one effect each. The idea is to use the specially constructed lifted rules of the extended Datalog program, ground them with a corresponding algorithm and use them as unary operators for calculating the heuristic values. This could significantly reduce the number of unary operators, which can lead to a more efficient calculation. Experiments show that the new implementation enables a more efficient calculation of the additive heuristic values for some domains.

We begin with the essential background to the content of this work. Then, we analyze the specific parts of the Fast Downward code that are particularly relevant for this work. Next, we examine the optimization idea in more detail, describe how the optimizations are implemented with the changes in the code and discuss the limitations of the new implementation. We then evaluate the new implementation with experiments. Finally, we conclude with a summary of the main points and a discussion of future work.

2
Background

This section provides the essential background for understanding the following content. First, the planning task and important related terms are defined. Then, the additive heuristic is presented. Next, Datalog programs are explained and finally the construction of Datalog rules is shown, which is the basis of the optimizations.

2.1 Planning Task

A planning task in STRIPS representation (Fikes and Nilsson, 1971) is a 4-tuple $\Pi = \langle V, I, G, A \rangle$:

- V is a set of binary state variables. They can be true or false.
- I is the initial state. This is the situation at the beginning of the planning. The initial state is a set of state variables which are true in the initial state.
- G is the goal. This is the situation, that we want to reach. The goal is a set of state variables which are true in the goal.
- A is the set of actions or operators, where each action $a \in A$ has a precondition list $pre(a) \subseteq V$, an add list $add(a) \subseteq V$, a delete list $del(a) \subseteq V$ and a non-negative cost $cost(a)$.

A state is a set of state variables and describes the properties of a situation in a planning task so that exactly these state variables are true. States, where all state variables of the goal are true, are goal states.

An action is applicable to state s if $pre(a) \subseteq s$. The successor of s with applicable action a is defined as $s[a] = (s \setminus del(a)) \cup add(a)$. A sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ is applicable to s_0 if there are states s_i for $i = 1, ..., n$ with action a_i applicable in s_{i-1}, and $s_i = s_{i-1}[a]$. The result of applying π to s is denoted by $s[\pi] = s_n$. A plan is a solution for a planning task and consists of a sequence of actions $\pi = \langle a_1, ..., a_n \rangle$, which is applicable to the initial state s_0 and leads from s_0 to a goal state. An optimal plan has minimum cost among all plans, whereby the cost of a plan is $\sum_{i=0}^{n} cost(a_i)$.

In this work, we also need a lifted representation of a planning task. In order to enable a switch between the representations, we define the term atom. An atom $P(\langle t_1, ..., t_n \rangle)$ consists of an n-ary predicate symbol P and a tuple $\langle t_1, ..., t_n \rangle$, where $t_1, ..., t_n$ can be objects or variables. When grounding an atom, the variables of the atom are replaced by objects, whereby there is a variable substitution or variable mapping $\sigma : \mathcal{V} \mapsto O$. Ground atoms are similar to state variables.

A planning task in lifted STRIPS representation similar to Corrêa et al. (2021b) but without inequality constraints is a 5-tuple $\Pi_{\text{lifted}} = \langle \mathcal{P}, O, \mathcal{A}, s_0, \gamma \rangle$:

- $\mathcal{P}$ is a set of predicate symbols.
- O is a set of objects, which are constants.
- $\mathcal{A}$ is the set of actions, where each action $a[\Delta] \in \mathcal{A}$ has a precondition list $pre(a[\Delta])$, an add list $add(a[\Delta])$, a delete list $del(a[\Delta])$ and a non-negative cost $cost(a[\Delta])$. Here, Δ is a set of variables. The three lists of an action consist of atoms.
- s_0 is the initial state, which is the situation at the beginning of the planning. The initial state is a set of ground atoms which are true.
- γ is the goal, which is the situation, that we want to reach. The goal is a set of ground atoms which are true.

2.2 Additive Heuristic

Heuristics help to focus the search on promising sequences of actions. They estimate the distance from the states to the nearest goal state. The heuristic considered in this paper is the additive heuristic h^{add}. It is originally defined on the ground planning task Π and looks at ground atoms. The additive heuristic is a delete relaxation heuristic (Bonet and Geffner, 2001). Delete relaxation heuristics compute their estimate on a simplification of the original task where delete lists are empty. This means that no atoms are removed when actions are applied, only new atoms are added. All atoms that can be reached in the planning task Π by a sequence of actions can also be reached in the relaxed planning task Π^+. The additive heuristic assigns each atom a cost that indicates how expensive it is to reach a state where this atom is true. These costs are computed by looking at all actions that reach the atom and summing the cost of their preconditions together, and then taking the minimum. The single value $h(p, s)$ is an estimate of the cost of reaching an atom p from state s in the relaxed planning task Π^+:

$$h(p,s) = \begin{cases} 0, & \text{if } p \in s, \\ \min\limits_{a \in A_p} \{cost(a) + \sum\limits_{q \in pre(a)} h(q,s)\}, & \text{otherwise}, \end{cases} \tag{2.1}$$

where A_p is the set of ground actions which have p in their add list. These values for each ground atom in the goal are added for h^{add} to obtain the value of the heuristic:

$$h^{\text{add}}(s) = \sum_{p \in \gamma} h(p,s). \tag{2.2}$$

However, the additive heuristic does not provide perfect estimates, i.e. the exact costs, for Π^+, as it assumes that reaching the atoms in the goal, i.e. reaching the subgoals, is independent. It is therefore not taken into account that reaching a subgoal can help to reach another subgoal.

2.3 Datalog

Datalog programs are logic programs. For the optimizations and for the grounding to obtain the operators for calculating the heuristic, weighted Datalog programs are used, as defined by Corrêa et al. (2021b). For the grounding of the actions in the original Fast Downward, Datalog programs without weights are used, as defined by Helmert (2009). A weighted Datalog program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ consists of a set of facts $\mathcal{F}$, which are ground atoms, and a set of weighted rules $\mathcal{R}$. A weighted Datalog rule r consists of atoms ϕ_i and is a first-order formula of the form $\phi_1 \wedge ... \wedge \phi_m \rightarrow \phi_0$, for $m \geq 0$. It can also be written in the form $\phi_0 \leftarrow \phi_1, ..., \phi_m$. Here, $head(r) = \phi_0$ is the head of the rule and $body(r) = \{\phi_1, ..., \phi_m\}$ is the body of the rule. The atoms can contain objects, but usually consist of variables. The rule has a non-negative weight $w(r)$, which is important in this work for calculating the cost of an operator and thus for the correct values of the additive heuristic.

Datalog programs help to calculate the set of reachable atoms. To do this, they calculate the set of ground atoms ϕ, which the Datalog program logically implies. This is the canonical model $\mathcal{M}$ of $\mathcal{D}$, so that $\mathcal{F} \cup \{r_\forall | r \in \mathcal{R}\} \models \phi$. Here, $vars(r) = \{v_1, ..., v_n\}$ are the variables of the rule r and $r_\forall \equiv \forall v_1 ... v_k \cdot \phi_1 \wedge ... \wedge \phi_m \rightarrow \phi_0$. To calculate $\mathcal{M}$, a specific Datalog program $\mathcal{D}(\Pi_{\text{lifted}}, s_0)$ is created for the planning task Π_{lifted}. The set $\mathcal{F}$ consists of all ground atoms of the initial state s_0 of the planning task. The set of rules $\mathcal{R}$ contains the following rules:

- For the goal $\gamma = \{\gamma_1, ..., \gamma_n\}$ there is a rule r of the form

$$\text{goal-reachable} \leftarrow \gamma_1, ..., \gamma_n, \tag{2.3}$$

  where $w(r) = 0$ if the rules are weighted.

- For each action $a[\Delta] \in \mathcal{A}$ with $pre(a[\Delta]) = \{\phi_1, ..., \phi_n\}$ there is an applicability rule r of the form

$$a\text{-applicable} \leftarrow \phi_1, ..., \phi_n, \tag{2.4}$$

  where $w(r) = cost(a)$ if the rules are weighted, and the a-applicable is the action predicate of a corresponding action.

- For each atom ψ in $add(a[\Delta])$ there is an effect rule r of the form

$$\psi \leftarrow a\text{-applicable}, \tag{2.5}$$

  where $w(r) = 0$ if the rules are weighted.

For the grounding of the actions and rules to obtain the operators, we need the canonical model $\mathcal{M}$ of $\mathcal{D}(\Pi_{\text{lifted}}, s_0)$. Helmert (2009) shows that $\mathcal{M}$ contains all ground atoms that are

reachable from s_0 in the delete relaxation Π^+, thus also the a-applicable of the applicable ground actions as well as the goal-reachable atom, if all goal atoms are reachable.

Corrêa et al. (2021b) show that the values of the additive heuristic can be calculated using the weighted Datalog programs.

2.4 Construction of Rules

The rules for the calculation of the additive heuristic, i.e. those of the weighted Datalog program, are constructed in a certain way to enable the weighted rules to be grounded to operators at all and to allow for efficient calculations. Three rule construction approaches are employed. These are the rule splitting by Helmert (2009), which was adapted for weighted rules by Corrêa et al. (2021b), and action predicate removal and duplicate rule removal by Corrêa et al. (2021b). These approaches will be explained in the order in which they should be used for the construction of the weighted rules.

2.4.1 Action Predicate Removal

The rules which have an a-applicable atom are changed here. This atom is removed from the respective rules. There are no more applicability rules (2.4). For each effect rule (2.5) of an action, the atoms of the precondition list of the action are set directly as the rule body instead of the a-applicable atom. These new rules therefore look like this:

$$\psi \leftarrow \phi_1, ..., \phi_n, \tag{2.6}$$

where $\{\phi_1, ..., \phi_n\}$ is the precondition of an action and ψ is an atom of the add list. The weight of such rules is the cost of the corresponding action.

2.4.2 Rule Splitting

This rule rewriting is used in the original Fast Downward to modify the rules of the Datalog program in such a way that the canonical model can be calculated efficiently, as efficient unification queries are made possible. Certain decompositions of the rules can improve performance, as special data structures can be used. The rules are split into smaller rules during rule splitting, and query optimization techniques are used to split the rules. First, it is checked that no variables occur more than once in the individual atoms of a rule, otherwise the duplicate variables are removed. Then, the individual rules are split until they only contain two atoms in the body. Each rule $r \equiv \phi_0 \leftarrow \phi_1, ..., \phi_i, ..., \phi_j, ..., \phi_m$ in $\mathcal{R}$ is replaced by two rules r_1 and r_2. This can result in two different forms:

$$r_1 \equiv \phi_0 \leftarrow \theta, \phi_1, ..., \phi_{i-1}, \phi_{i+1}, ..., \phi_m$$
$$r_2 \equiv \theta \leftarrow \phi_i$$

or

$$r_1 \equiv \phi_0 \leftarrow \theta, \phi_1, ..., \phi_{i-1}, \phi_{i+1}, ..., \phi_{j-1}, \phi_{j+1}, ..., \phi_m$$
$$r_2 \equiv \theta \leftarrow \phi_i, \phi_j.$$

Here, θ is a new auxiliary atom. For weighted rules, the weights of the new rules are $w(r_1) = w(r)$ and $w(r_2) = 0$.

2.4.3 Duplicate Rule Removal

This optimization removes rules that are only different due to the naming of the variables. In rule splitting, many such rules are created that have an auxiliary predicate as effect. Only one of these syntactically equivalent rules is kept. The costs are not affected, as this type of rule that defines an auxiliary predicate has weight 0.

3
Fast Downward

This work explores how the optimizations for the calculation of the additive heuristic can be integrated into Fast Downward. Fast Downward is a state-of-the-art classical planning system that works on a ground representation (Helmert, 2006). In this work, we focus on certain parts of the code of Fast Downward. There is a division into two components: translate and search. These two components of the original Fast Downward are examined in more detail in the following.

3.1 Translate

The translate component is responsible for translating planning tasks formulated in PDDL into the finite-domain representation (FDR) (Helmert, 2009). PDDL is a lifted representation, while FDR is a grounded representation that uses multi-valued variables. In this component happens the grounding of the planning task. The code is written in Python. The following parts of the translate code are particularly important for this work:

PDDL to FDR Task The `translate.py` file contains the function `pddl_to_sas`, which is called in the `main` function. This function takes the normalized lifted planning task, which is a task with specific structural restrictions, and creates a ground FDR task through a series of steps. Following additional analysis in this function, the number of reachable atoms is reduced. Among other things, the FDR task contains variables whose respective possible values are a certain group of the final reachable atoms. These groups of atoms of the variables are disjoint. Moreover, the FDR task contains the FDR operators that use these variables. The FDR operators were originally derived from the ground actions, which have undergone some processing. The FDR task is written to an output file in a specific format at the end of the function.

At the very beginning of `pddl_to_sas`, the `explore` function of the `instantiate.py` file is called. This function calls further functions and returns the instantiated, i.e. grounded, planning task. The functions called there are outlined briefly below:

Translating PDDL to Logic Program The first function called is `translate` in the file `pddl_to_prolog.py`. Here, the Datalog program of the planning task is created and rule splitting is applied.

Building Canonical Model The second function called is `compute_model` in the file `build_model.py`. It takes the created Datalog program and computes the corresponding canonical model that contains all reachable atoms.

Grounding of Planning Task The third function called is `instantiate`, which, like `explore`, is located in `instantiate.py`. This function takes the planning task and the canonical model, uses the reachable atoms for instantiation and returns, among other things, the grounded actions.

3.2 Search

The search component is responsible for finding a plan for the ground planning task, which is parsed from the output file of the translate component. The code is written in C++.

The files `relaxation_heuristic.cc` and `additive_heuristic.cc` with the corresponding header files are important for calculating the values of the additive heuristic. The `AdditiveHeuristic` class is derived from the `RelaxationHeuristic` class and has functions for calculating the heuristic values. `RelaxationHeuristic` is responsible for creating the data structures for the propositions and unary operators, which are explained in the next paragraph.

In Fast Downward, relaxation heuristics are computed in the following way. The ground actions are split into so-called unary operators, this means that for each effect a unary operator is defined. Furthermore, we use a binary representation of the variables called propositions, such that each variable-value pair of an FDR variable is a proposition. For computing the heuristic, a priority queue of propositions and their cost is maintained. Initially, all propositions which are true in the state are inserted with cost 0. When a proposition is removed, we check if new unary operators can now be applied and if so, we insert its effect with the appropriate cost into the queue.

The propositions are stored in a vector called `propositions`. They have an associated proposition ID, which is the index of the proposition in the vector. The ID can be determined using the variable and the value of the fact. The unary operators are stored in a vector called `unary_operators`. The data structures are built in the constructor of the `RelaxationHeuristic` class. The `build_unary_operators` function is called there for each operator. It takes the operator and builds the associated unary operators, each of which has only one effect.

4

Changes with Applied Optimizations

The aim of this chapter is to examine the optimization idea in more detail and to present the changes and adjustments to the Fast Downward planning system that are necessary for the implementation of the optimizations.

4.1 Optimization Idea

We can reformulate unary operators in different ways as long as we ensure that they represent the same relaxed task. Different formulations might lead to faster computations if the number of unary operators is smaller. The weighted Datalog program with the construction of the rules as in Section 2.4 is such a reformulation where we also have unary operators, i.e. operators with one effect, but in a lifted representation. Using this weighted Datalog program instead of using the actions for the unary operators can reduce the number of unary operators and simplify their structure. Therefore, if we directly use the optimized rules for the unary operators, it can allow a more efficient computation of the values of the additive heuristic. To illustrate the differences and the construction of rules, let us compare actions and rules and the number of ground unary operators in an example:

Actions Let $a[\Delta]$ be an action with $pre(a[\Delta]) = \{\mathrm{P}(x), \mathrm{Q}(x,y), \mathrm{R}(z)\}$, $add(a[\Delta]) = \{\mathrm{A}(x), \mathrm{B}(y)\}$, $del(a[\Delta]) = \emptyset$ and with $cost(a[\Delta]) = 1$, where $\Delta = \{x, y, z\}$. Let $O = \{o_1, o_2, o_3\}$ be the set of objects. If each variable of Δ can be mapped to each object of O, after grounding we get $|O|^{|\Delta|} \cdot |add(a[\Delta])| = 3^3 \cdot 2 = 54$ unary operators. A unary operator consists of one ground atom from the add list and all ground atoms from the precondition list.

Rules The rules corresponding to the action without the rule construction approaches are the effect rules

$$\mathrm{A}(x) \leftarrow a\text{-applicable},$$
$$\mathrm{B}(y) \leftarrow a\text{-applicable},$$

where the weight of each rule is 0, and the applicability rule

$$a\text{-applicable} \leftarrow \mathrm{P}(x), \mathrm{Q}(x, y), \mathrm{R}(z),$$

with weight 1. If we remove the action predicate, we get

$$\mathrm{A}(x) \leftarrow \mathrm{P}(x), \mathrm{Q}(x, y), \mathrm{R}(z),$$
$$\mathrm{B}(y) \leftarrow \mathrm{P}(x), \mathrm{Q}(x, y), \mathrm{R}(z),$$

where the weight of each rule is 1. If we apply the rule splitting, we get

$$\mathrm{A}(x) \leftarrow \theta_0(x), \theta_1(),$$
$$\mathrm{B}(y) \leftarrow \theta_4(y), \theta_5(),$$

each with weight 1 and

$$\theta_2(x) \leftarrow \mathrm{Q}(x, y),$$
$$\theta_0(x) \leftarrow \theta_2(x), \mathrm{P}(x),$$
$$\theta_1() \leftarrow \mathrm{R}(z),$$
$$\theta_4(y) \leftarrow \mathrm{Q}(x, y), \mathrm{P}(x),$$
$$\theta_5() \leftarrow \mathrm{R}(z),$$

where the weight of each rule is 0. In the rule splitting, some created auxiliary atoms are removed again, which is why there is no atom with predicate θ_3 in this example. Next, we remove the duplicate rule $\theta_5() \leftarrow \mathrm{R}(z)$ and leave the rules

$$\mathrm{A}(x) \leftarrow \theta_0(x), \theta_1(),$$
$$\mathrm{B}(y) \leftarrow \theta_4(y), \theta_1(),$$

each with weight 1 and

$$\theta_2(x) \leftarrow \mathrm{Q}(x, y),$$
$$\theta_0(x) \leftarrow \theta_2(x), \mathrm{P}(x),$$
$$\theta_1() \leftarrow \mathrm{R}(z),$$
$$\theta_4(y) \leftarrow \mathrm{Q}(x, y), \mathrm{P}(x),$$

each with weight 0. This results in six rules. Only the variables that are important for the add effect remain, so four rules depend on one variable and two rules depend on two variables. If each variable can be mapped to any object, we get $3^1 \cdot 4 + 3^2 \cdot 2 = 30$ unary operators after grounding the rules. Rules inherently have a form corresponding to the form of unary operators, i.e. with only one add effect.

4.2 Changes in Translate

To implement the optimization idea in Fast Downward, the translate component has to be extended with new code. This includes the building of a second Datalog program with the

special construction of the rules and a corresponding canonical model, a separate grounding algorithm for the rules and the writing of the unary operators into another output file.

After calling the function `pddl_to_sas` in the main function of `translate.py` we add another function call of a new function `compute_operators_for_relaxation_heuristic` in `instantiate.py`. This is the higher-level function. It takes the PDDL planning task and the FDR task as arguments. First, it calls a new function `translate_optimize` of the `pddl_to_prolog.py` file to get the Datalog program. Then, it calls the function `compute_model` of the `build_model.py` file with the result of the previously called function to compute the canonical model. Next, it builds a dictionary with the predicates of the atoms of the FDR task as key and the list of matching atoms as value. The dictionary helps to find an atom quickly. To get the atoms, it uses the values of the variables of the FDR task. After that, it calls the new function `instantiate_for_relaxation_heuristic`, which takes the Datalog program, the canonical model, the dictionary with the FDR task atoms and the facts that are true in the initial state in the PDDL and instantiates the rules to get the ground unary operators. Finally, it calls the new function `output` to write the unary operators to a new output file. In the following subsections, we will examine the called functions.

4.2.1 Rule Construction Code

The function `translate_optimize` has the same basic structure as the original `translate` but also contains the important additional function calls that are required for the optimizations. First, the function `translate_optimize` calls the function `remove_duplicate_preconditions_in_actions`. It is theoretically possible for an action in a PDDL domain to have duplicate preconditions. However, this is an extremely rare occurrence. For example, in the action `take_image` of the domain Satellite, there is the precondition `power_on ?i` twice. This can result in erroneous, higher heuristic values due to the repeated application of rules with a weight equal to the action costs. To handle these cases, the duplicates are removed in this function if the precondition is a conjunction. Next, the Datalog program is generated in `translate_optimize`. Before normalizing it, we use the function `remove_action_predicates`. The code for constructing the rules was taken from Corrêa et al. (2021a) and only slightly modified. This applies in particular to the functions `remove_action_predicates`, `rename_free_variables` and `remove_duplicated_rules`, as well as the changes for weighted rules in the file `split_rules.py`. Following the removal of action predicates, the `normalize` function is called. However, a property of normalized Datalog programs is ignored here: rules with an empty precondition are allowed. In our case, this type of rule is necessary for correct functioning, for example to find a solution for planning tasks of the domain Movie. The add effects of such rules are also added to the facts of the Datalog program. Next, the three functions `split_rules`, `rename_free_variables` and `remove_duplicated_rules` are called in `translate_optimize` to rewrite the rules. Finally, the created Datalog program is returned.

4.2.2 Grounding of Rules

In the function `instantiate_for_relaxation_heuristic`, we first create two dictionaries for the canonical model and for the facts that are true in the initial state in the PDDL. These dictionaries are similar to the dictionary for the values of the FDR variables with the predicate of the atom as key and the list of matching atoms as value. This makes finding an atom faster than in a simple list or set. Next, each rule is grounded individually. The goal rule is skipped, as it is only needed for the calculation of the canonical model. We call the `instantiate_rule` function, which takes as arguments the facts of the initial state, an empty dictionary for the variable mapping, the preconditions of the rule, the effect of the rule, the canonical model, the values of the FDR variables and an empty list called atoms. This function returns the ground unary operators. Each unary operator is appended to a list of ground operators as a tuple consisting of the effect, a tuple of the preconditions and the weight of the rule, which is now the cost of the operator. When all rules have been grounded, this list is returned as a set to remove duplicate operators.

Let us now examine the function `instantiate_rule`. The function is recursive, with each call shortening the list of preconditions, possibly extending the variable mapping and checking whether a unary operator can be formed from the atoms.

- *Base Case:* The base case is reached if the list of preconditions is empty. In this case, the preconditions have already been checked and grounded, if preconditions exist, and the effect is now processed here. First we check whether the predicate of the atom is contained in the canonical model at all. If not, an empty list is returned. If the predicate is in the model, for each of these atoms containing the predicate of the effect in the model, we check whether it can be the effect of an operator. The aim here is to determine whether the atom of the model matches both the effect atom and the variable mapping of the preconditions of the rule. First it has to be true that it is either an auxiliary atom or is contained in the values of the FDR variables, so this atom is not removed by the original implementation. If this is the case, each argument in the tuple of the effect atom is checked. If the argument is a variable, we check in the variable mapping whether the value of the variable matches the value of the argument in the ground atom of the model. If it is an object, which means that the effect is already partially grounded, as it happens for example with the domain Airport, we compare it directly with the argument. When all arguments are matched, we append a tuple consisting of the atom from the model as the effect and the list of ground precondition atoms to a list of ground operators. After each atom in the model containing the predicate of the effect has been tested, the list of ground operators is returned.

- *Recursive Case:* If the list of preconditions is not empty, the first condition in this list is set as the current condition, which is now examined. First, we check whether the predicate of the atom is contained in the canonical model at all. If not, an empty list is returned, which means that the rule cannot be grounded. If the predicate is in the model, we test for each of these atoms containing the predicate of the condition

in the model whether it can be a precondition of an operator. The aim here is to determine whether the atom of the model matches both the condition atom and the variable mapping of previously grounded conditions of the rule. We use a copy of the variable mapping to create separate mappings for each matching atom of the model. Now we inspect each argument in the tuple of the condition atom. If the argument is a variable, we check with the variable mapping whether the value of the variable has already been set. If not, the variable is assigned to the value of the argument. If it has already been set, it is checked whether the value of the variable matches the value of the argument. If the argument is an object, we compare it directly with the argument of the ground atom of the model. When all arguments are matched, we distinguish between different cases:

- In the first case, it is ensured that the atom in question is not a special atom with an @ symbol that is only required for the generation of the canonical model. In addition, it must be either an auxiliary atom or a value of the FDR variables. If these conditions are met, the `instantiate_rule` function is called recursively with the following changes: the variable mapping is possibly extended, the list of preconditions is shortened and the list of atoms is extended by the ground atom of the precondition.
- If the atom is neither auxiliary or an FDR fact and it is not an atom with an @ symbol, the translator must have removed it during its processing. This can happen for two reasons which can be easily distinguished by checking whether the atom is contained in the initial state described in the PDDL:
  a) If the atom is contained in the initial state, the atom is true in every reachable state. Here, the `instantiate_rule` function is called recursively, whereby the variable mapping is possibly extended, the list of preconditions is shortened and the list of atoms is not extended by the ground atom of the precondition. We ignore this atom and it will not appear in the preconditions of the operator. In this case we also handle atoms with an @ symbol, which can be ignored.
  b) If the atom is not contained in the initial state, the atom is unreachable, which means it is false in every reachable state. Consequently, there can be no such operator.

The list of ground unary operators that is returned when `instantiate_rule` is called recursively is appended to a result, which in turn is returned. This allows for the collection of all ground operators of a rule in a single result.

4.2.3 New Output File

We write the ground unary operators in the function `output` to a new output file `operators_relaxation_heuristic.txt`. Here we process the operators one by one. First, we write the effect atom on one line. Then, we write each precondition atom on a separate line. After the preconditions, the string `cost` is written on a separate line and the

cost number is set on the next line. When all the operators have been processed, we end the output with the string `end_operators`.

4.3 Changes in Search: Constructor of RelaxationHeuristic

In order to use the new unary operators, the code of the `relaxation_heuristic.cc` file and the corresponding header file has to be changed in the search component of Fast Downward. This mainly affects the constructor of the `RelaxationHeuristic` class with the functions called there. A new function `parse_operators` as well as new structs `ParsedOperator` and `ParsedProposition` with corresponding vectors `parsed_operators` and `new_propositions` were added and the function `build_unary_operators` was changed. We will now examine the changes to the constructor more closely.

In the constructor of the `RelaxationHeuristic` class, we first calculate the number of facts. This is the number of values of the FDR variables. Then we call the `parse_operators` function to get the unary operators from the new output file and insert them into the `parsed_operators` vector. After proposition offsets are formed as in the original constructor, we now build the unary operators. To achieve this, we use the `parsed_operators` vector instead of the FDR operators and call the `build_unary_operators` function for each parsed operator. Only then do we build the vector `propositions`, whose size here is the sum of the number of facts and the number of new propositions, which is the size of the vector `new_propositions` that is set during building the unary operators. Next, we handle goal propositions like in the original implementation and remember their proposition IDs. The rest of the constructor remains as in the original implementation. The following subsections will examine the new `parse_operators` function and the modified `build_unary_operators` function in greater detail.

4.3.1 Parsing Output File

In the function `parse_operators` we open the output file with the unary operators `operators_relaxation_heuristic.txt`. As long as the content of the parsed line is not the string `end_operators`, i.e. there are more operators to parse, we create a new `ParsedOperator` with a `ParsedProposition effect` and a vector `precondition` with `ParsedPropositions`. We first parse the effect atom. This will be the name of the `ParsedProposition effect`. Now as long as the content of the next line is not the string `cost`, the content is a precondition atom. We set the atom as the name of a `ParsedProposition`, which is inserted into the `precondition` vector. If the content of the next line is the string `cost`, all preconditions have been parsed. We parse the cost number and set it as the cost of the operator. At this point, the full unary operator has been parsed and the `ParsedOperator` is inserted into the `parsed_operators` vector.

4.3.2 Building Unary Operators

The function `build_unary_operators` is called for each `ParsedOperator` with the `ParsedOperator` as argument. Initially, the vector `precondition_props` is constructed, which will contain the proposition IDs of the preconditions. Each `ParsedProposition` of the preconditions of the `ParsedOperator` is then processed. First, we examine the facts of the task to determine whether the atom of the precondition is contained there. If a matching fact is found, the proposition ID of the fact is inserted into the `precondition_props` vector. If no such atom was found in the facts, we use a new function `get_prop_id` that checks the proposition ID of a `ParsedProposition` and returns or sets the value. If the proposition ID of the precondition has not yet been set, it is now set. It is then inserted into the `precondition_props` vector. Once all preconditions have been processed, a similar process is repeated with the effect of the `ParsedOperator`. When the unary operator is finally built in the `unary_operators` vector, we use the value `NO_OP` for the index to the matching FDR operator of the task, as we do not have such an index. The cost of the `ParsedOperator` is taken as the base cost of the unary operator.

We will now examine the new `get_prop_id` function, which was previously mentioned. The original implementation already has `get_prop_id` functions with different arguments. In `build_unary_operators` we additionally use a new function `get_prop_id` with a `ParsedProposition prop` and a boolean `set_id` as arguments. This function should only be called for propositions that are not contained in the task's facts. In the function, we first search for the proposition in the vector `new_propositions`. If the proposition is found, we set the ID of `prop` to the value of the ID of the proposition in the vector and return it. If the proposition is not contained in the vector, we check the value of `set_id`. If the value is true, we set a new ID. To do this, we use the sum of the number of facts and the size of the `new_propositions` vector. Then we insert the proposition into the vector and return the ID. If the value of `set_id` is false, we return the value -1. This means that the proposition was not found and the ID is not set.

4.4 Limitations

Using the specially constructed rules instead of the FDR operators for the unary operators leads to certain limitations, which are discussed in more detail in this section.

- **Cost Depending on Parameter**: In a PDDL planning task, there may be action costs that depend on parameters. This means that the costs can only be determined when the action is grounded, because the objects for the parameters are set there. Since we do not know which ground action corresponds to which ground rules, we cannot determine such costs. Therefore they are not supported. This applies for example to the domains Agricola, Data-Network, Elevators, Transport and Woodworking.
- **Negative Preconditions**: PDDL actions can have negative preconditions. They are compiled away in the FDR representation. To compile them away in the new implementation, we need a mapping from FDR operators to ground rules. Since we

do not have such a mapping, we cannot handle these cases. This applies for example to the domains Organic-Synthesis, Quantum-Layout, Snake, Spider, Termes, Tetris and Tidybot.

- **Removal of Inapplicable Operators**: Since Fast Downward removes inapplicable operators after grounding the actions, such as with the help of mutex information, the initial heuristic values may be different from this implementation, which uses ground rules as unary operators and does not remove these inapplicable operators. This is not easily transferable to the implementation in this work. This means that in some cases the task is not as well optimized as in the original implementation. This happens for example in the domains of Freecell, Pipesworld-Tankage and Scanalyzer.

  To illustrate, consider the Scanalyzer domain. It describes a planning task that has to do with the analysis of cars on segments. These cars are moved in circles of segments and can either be analyzed or simply rotated. Let us examine certain elements of the domain in the PDDL formulation:

  There are two types: segment, which is a part of the path on which cars are moved, and car, that can be analyzed and moved. The action analyze-2 has the parameters ?s1 and ?s2, which are segments, and ?c1 and ?c2, which are cars. There are three preconditions:

  (CYCLE-2-WITH-ANALYSIS ?s1 ?s2 - segment): The segments ?s1 and ?s2 form a 2-cycle where analysis is possible.

  (on ?c1 ?s1): The car ?c1 is on the segment ?s1.

  (on ?c2 ?s2): The car ?c2 is on the segment ?s2.

  The effects are the following:

  (not (on ?c1 ?s1)): Car ?c1 is not on ?s1.

  (not (on ?c2 ?s2)): Car ?c2 is not on ?s2.

  (on ?c1 ?s2): Car ?c1 is on ?s2.

  (on ?c2 ?s1): Car ?c2 is on ?s1.

  (analyzed ?c1): Marks ?c1 as analyzed.

  (increase (total-cost) 3): Increases the total cost by 3.

  The action rotate-2 has the same parameters as analyze-2. There are three preconditions:

  (CYCLE-2 ?s1 ?s2): The segments ?s1 and ?s2 form a 2-cycle.

  (on ?c1 ?s1): The car ?c1 is on the segment ?s1.

  (on ?c2 ?s2)The car ?c2 is on the segment ?s2.

  The effects are:

(not (on ?c1 ?s1)): Car ?c1 is not on ?s1.

(not (on ?c2 ?s2)): Car ?c2 is not on ?s2.

(on ?c1 ?s2): Car ?c1 is on ?s2.

(on ?c2 ?s1): Car ?c2 is on ?s1.

(increase (total-cost) 1): Increases the total cost by 1.

Consider the proposition analyzed(car-out-1) of one of the tasks in the domain. In this task, we have the cars car-in-1, car-in-2, car-in-3, car-out-1, car-out-2, car-out-3 and the segments seg-in-1, seg-in-2, seg-in-3, seg-out-1, seg-out-2, seg-out-3. We set co1 for car-out-1, ci1 for car-in-1, so1 for seg-out-1, si1 for seg-in-1, and similarly for the others. In the following, the propositions which contain p$ are auxiliary atoms. The value obtained for the proposition analyzed(car-out-1) is 4, whereas the original implementation yields a value of 5 for this proposition. Figure 4.1 shows the rules that can be used to reach the proposition. For instance, the proposition analyzed(co1) can be reached via the propositions on(co1, so1), on(co1, si1) and p$0(si1, so1). This would correspond to the inapplicable action analyze-2 with seg-out-1, seg-in-1, car-out-1 and car-out-1, which would mean that car-out-1 is located on two different segments at the same time. By employing such rules, it is possible to circumvent the rules associated with the action rotate-2. Therefore, only cost 1 for reaching the proposition on(co1, si1), which is associated with the action rotate-2, is added to cost 3 for reaching the proposition with the predicate analyze, which is associated with action analyze-2. Therefore, we get a cost of 4.

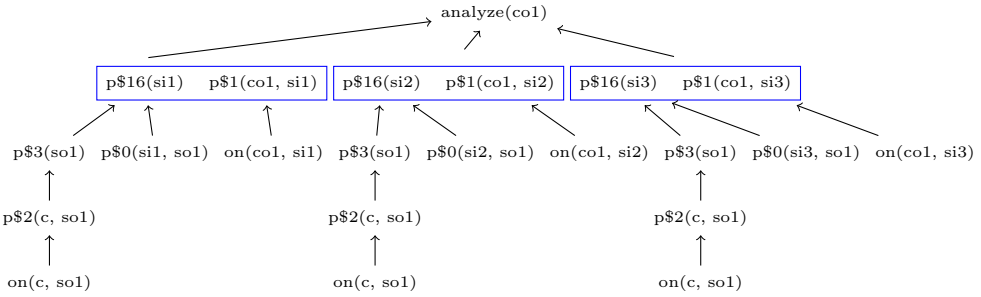


Figure 4.1: Graph of the rules to reach the proposition analyze(car-out-1). The blue boxes are different groups of preconditions. To reach the proposition, we only need one of these groups. In p$2(c, so1) there can be each car for c and then the precondition on(c, so1) has the same car c.

The original implementation has a cost of 1 for on(ci1, so1). This proposition is needed to use action analyse2 with car co1 on segment si1 and car ci1 on segment so1. Getting on(ci1, so1) is achieved with the same action that achieves on(co1, si1), but it is counted separately because we just add costs of preconditions. So we get a cost of 2 for these propositions and add a cost of 3 for the use of action analyze-2. This results in a cost of 5.

This utilization of the rules can be regarded as a kind of unauthorized short cut, as it corresponds to the use of an action that is not applicable. Since we use a different definition of the task with the ground rules as unary operators, the values are not incorrect, as the values of the heuristics are calculated correctly with this definition. However, this definition results in a less optimized task compared to the original implementation definition.

- **Preferred Operators**: The use of preferred operators can improve performance. However, we cannot use preferred operators because of the absence of a mapping from the new unary operators to the FDR operators.
- **Axioms**: This work did not examine the use of axioms.

5
Experimental Results

This chapter presents the results of the evaluation and examines the properties of domains that affect the new implementation.

5.1 Setup

The experiments were conducted using the Python package Downward Lab (Seipp et al., 2017) and executed on the infai nodes of the sciCORE scientific computing center at the University of Basel. The total duration of each run was constrained to 30 minutes, while the overall memory limit was set to 4GB. The implementations were evaluated on the suite satisficing STRIPS with IPC benchmarks up to and including IPC 23 [1]. Some benchmarks had to be omitted due to some limitations of the new implementation, as detailed in Section 4.4. The heuristic used is, as previously discussed, h^{add} and the search algorithm used is eager best-first search. The following metrics are examined to compare the new implementation with the original implementation:

Coverage: This is the number of planning tasks for which a solution was found within the specified time limit and memory limit.

Memory: This is the raw memory used by the planner.

Planner Time: This refers to the time needed for the translation part, the preparation of the search, including the construction of unary operators, and the search itself.

Search Time: This is the time used for the search.

Total Time: This is the time used for the preparation of the search, including the construction of unary operators, and the search itself.

Unary Operators: This is the number of unary operators that are built for the calculation of the heuristic values.

[1] https://github.com/aibasel/downward-benchmarks

5.2 Results

The results of the experiments are presented in Table 5.1 and visualized in Figure 5.1. For both implementations, the coverage summed over all domains is identical. The new implementation uses slightly more memory. Despite the assumption that the new implementation would improve the search time in general, this is not the case, although the number of unary operators is significantly reduced with the new implementation. The total time and the planner time are generally much better with the original implementation. Figure 5.1 illustrates that, for the majority of planning tasks, the total time and the planner time are both longer with the new implementation. However, there are tasks for which the search time is significantly reduced, resulting in a positive effect on the total time and the planner time.

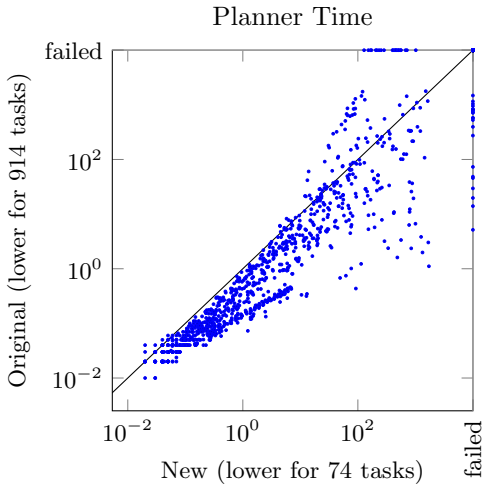
Summary	New	Original
Coverage - Sum	**1'030**	**1'030**
Memory - Sum	48'147'112	**47'563'024**
Planner time - Geometric mean	3.51	**1.42**
Search time - Geometric mean	0.60	**0.45**
Total time - Geometric mean	1.00	**0.53**
Unary operators - Sum	**12'112'234**	42'903'919

Table 5.1: Results for the metrics for the new implementation and the original implementation. The better values are presented in bold font.

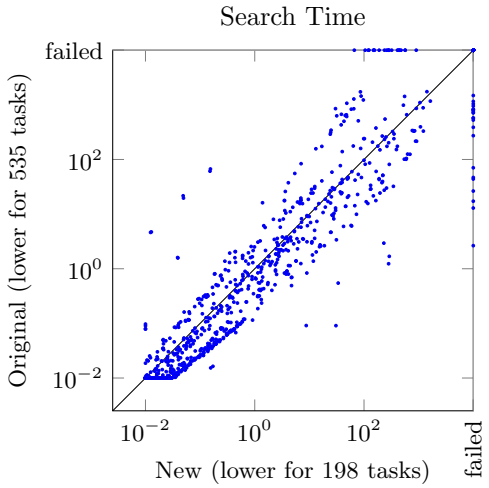
The question arises as to why the new implementation is slower in the search, despite a reduction in the number of unary operators. One potential explanation for the observed increase in search time is that domains with a different number of evaluations in both implementations may be contributing to this effect. These domains are Freecell, Pipesworld-tankage, Scanalyzer-08-strips and Scanalyzer-sat11-strips. Table 5.2 presents the results of the experiments, but considers only the domains with the same number of evaluations. However, the assumption is not confirmed, since the general search time is longer without these domains. It is evident that these domains are responsible for a considerable number of created unary operators in the original implementation.

Summary	New	Original
Coverage - Sum	**880**	879
Memory - Sum	42'812'408	**41'582'716**
Planner time - Geometric mean	3.37	**1.30**
Search time - Geometric mean	0.67	**0.45**
Total time - Geometric mean	1.03	**0.51**
Unary operators - Sum	**10'427'616**	28'345'981

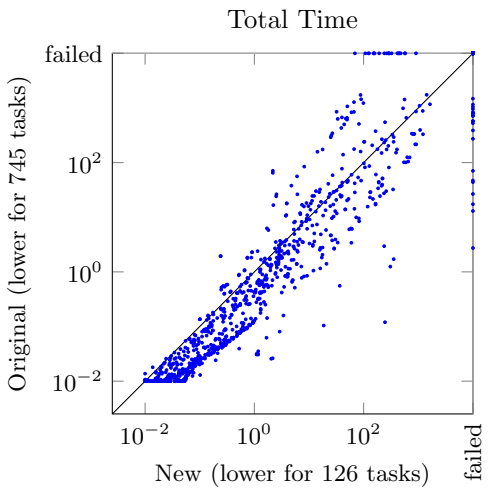
Table 5.2: Results for the metrics for the new implementation and the original implementation. Here, only domains with the same number of evaluations in both implementations are considered. The better values are presented in bold font.



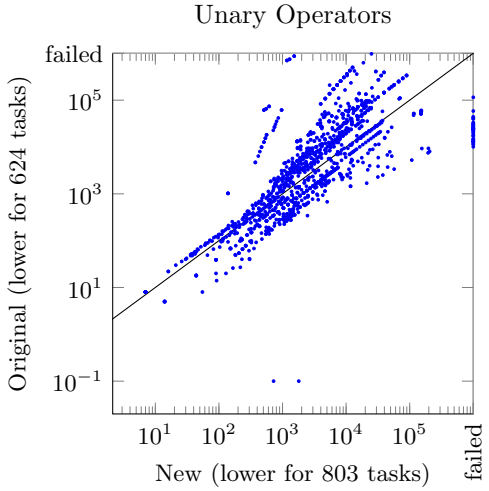
(a) Planner time of the implementations per planning task.



(b) Search time of the implementations per planning task.



(c) Total time of the implementations per planning task.



(d) Number of unary operators of the implementations per planning task.

Figure 5.1: Scatter plots for visualization of the individual metrics per planning task and for comparison of the new implementation with the original implementation.

Table 5.3 presents the coverage for domains. Only the domains for which the coverage differs in the two implementations are included in the table. It can be observed that the new implementation provides better coverage for the domains Parking-sat11-strips, Parking-sat14-strips and Satellite. For the Logistics98 domain, the coverage is significantly lower with the new implementation.

Coverage	New	Original
driverlog (20)	18	**19**
logistics98 (35)	18	**27**
parcprinter-08-strips (30)	23	**24**
parking-sat11-strips (20)	**20**	18
parking-sat14-strips (20)	**20**	5
pipesworld-notankage (50)	26	**27**
pipesworld-tankage (50)	20	**21**
rovers (40)	25	**29**
satellite (36)	**34**	30
storage (30)	16	**17**
thoughtful-sat14-strips (20)	12	**15**
Sum (1'466)	**1'030**	**1'030**

Table 5.3: Coverage of the new implementation and the original implementation in domains with different values of coverage. The better values are presented in bold font. The number of planning tasks in the domain is shown in parentheses.

5.3 Impact of Domain Properties

As demonstrated by the experiments, the new implementation affects different domains in different ways. This section analyzes the potential positive and negative properties of domains with respect to the new implementation.

Beneficial Domain Properties Since the coverage for the domain Parking-sat14-strips is significantly better with the new implementation, we investigate which properties of this domain might be beneficial, so that the heuristic values are computed more efficiently. For the domain Parking-sat14-strips, a total of 189'766 unary operators are built across all planning tasks with the new implementation and 6'527'262 unary operators with the original implementation. The PDDL formulation of the domain contains four actions, each with three variables. The actions have three or four preconditions, of which one precondition always has two variables, and they have two or three add effects with one or two variables. When constructing the rules, there are not so many rules that have two variables in the preconditions. In addition, many rules have the same structure, which allows for the removal of a significant number of duplicates.

Detrimental Domain Properties Coverage is significantly lower for the Logistics98 domain with the new implementation, so we investigate which properties of this domain may have a negative impact. For the Logistics98 domain, a total of 356'083 unary operators

are built across all planning tasks with the new implementation and 55'334 unary operators with the original implementation. The PDDL formulation of the domain contains six actions, each with three or four variables. The actions have four to seven preconditions, of which two preconditions always have two variables. Each action has only one add effect with two variables. When constructing the rules, many rules are created with two or three variables in the preconditions. This even applies to the majority of rules.

A comparison of the properties of the two domains reveals that the new implementation may be particularly effective for domains with actions that have a higher number of effects, each with preferably only one variable, and a lower number of preconditions, each with preferably only one variable. In this way, rules with few variables are more likely to be created, i.e. the rules are split in such a way that many variables are omitted. This can also result in many rules with the same structure, so that many duplicates are removed. The minimization of variables in rules leads to a reduction in the number of unary operators constructed during the grounding process.

6
Conclusion

In this thesis, optimizations were implemented to calculate the values of the additive heuristic in the classical planning system Fast Downward. This includes the use of specific weighted Datalog programs with specially constructed rules, an algorithm for grounding the rules and the use of these ground rules as unary operators. These optimizations can significantly reduce the number of unary operators used to calculate the values of the additive heuristic.

However, experiments have shown that although the overall number of unary operators is significantly reduced with the new implementation, it does not generally improve the search time. Nevertheless, in certain domains with specific properties, the new implementation improves performance and reduces the search time and in some cases even the entire planner time.

A significant issue exists in the absence of a mapping between the ground rules and the FDR operators. The implementation of such a mapping is complicated by the construction of the rules, especially by the duplicate rule removal and the variable renaming. One potential solution to obtain a mapping between the ground rules and the FDR operators would be to employ a similar approach to annotated Datalog programs as presented by Corrêa et al. (2022). Such an approach could facilitate the storage of transformations of the Datalog program, which could simplify the mapping.

Bibliography

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

Corrêa, A. B., Francès, G., Pommerening, F., and Helmert, M. (2021a). Code from the paper "Delete-Relaxation Heuristics for Lifted Classical Planning". https://doi.org/10.5281/zenodo.4594669.

Corrêa, A. B., Francès, G., Pommerening, F., and Helmert, M. (2021b). Delete-relaxation heuristics for lifted classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 94–102.

Corrêa, A. B., Pommerening, F., Helmert, M., and Francès, G. (2022). The FF heuristic for lifted classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 9716–9723.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208.

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

Helmert, M. (2009). Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5):503–535.

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

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:

Name Assessor: _____

Name Student: _____

Matriculation No.: _____

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: _____ Student: S. Wittner

Will this work, or parts of it, be published?

No

Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: _____ Student: _____

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis.