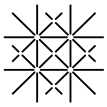

University of Basel

Finding Small Counterexamples of Expected Planner Behavior with Hill-Climbing

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
Supervisors: Dr. Gabriele Röger, Augusto B. Corrêa

Lucas Galery Käser
lucas.galerykaeser@gmail.com
15-051-618

7 October 2020

Acknowledgments

First and foremost, I would like to thank Dr. Gabriele Röger and Augusto B. Corrêa for their extraordinary support and guidance they offered as supervisors. I was always able to count on their help and answers to my endless questions.
Moreover, I would like to thank Prof. Dr. Malte Helmert for the opportunity to do my thesis in the Artificial Intelligence research group.
I also want to thank Salome Müller for helping me through frustrating times and for believing in me.
Finally, I want to thank my family and friends for their support.

Abstract

In the Automated Planning field, algorithms and systems are developed for exploring state spaces and ultimately finding an action sequence leading from a task's initial state to its goal. Such planning systems may sometimes show unexpected behavior, caused by a planning task or a bug in the planner itself. Generally speaking, finding the source of a bug tends to be easier when the cause can be isolated or simplified. In this thesis, we tackle this problem by making PDDL and SAS^+ tasks smaller while ensuring they still invoke a certain characteristic when executed with a planner. We implement a system that successively removes elements, such as objects, from a task and checks whether the transformed task still fails on the planner. Elements are removed in a syntactically consistent way, however, no semantic integrity is enforced. Our system's design is centered around the Fast Downward Planning System, as we re-use some of its translator modules and all test runs are performed with Fast Downward. At the core of our system, first-choice hill-climbing is used for optimization. Our "minimizer" takes (1) a failing planner execution command, (2) a description of the failing characteristic and (3) the type of element to be deleted as arguments. We evaluate our system's functionality on the basis of three use-cases. In our most successful test runs, (1) a SAS^+ task with initially 1536 operators and 184 variables is reduced to 2 operators and 2 variables and (2) a PDDL task with initially 46 actions, 62 objects and 29 predicate symbols is reduced to 2 actions, 6 objects and 4 predicates.

Table of Contents

1 Introduction

For researchers and developers in the Automated Planning field, finding the source of unexpected behavior in a planning system can be a tedious task. In this thesis, we develop an approach for reducing the size of a planning task causing some specific behavior on a planner while making sure the smaller instance still triggers the same behavior. Unlike the domain transformation method action schema splitting (Areces et al. 2014), we do not preserve task semantics in our approach. Our system (later referred to as *minimizer*) makes the task producing the bug smaller by removing one randomly selected task element, e.g., an action. The planner then is executed with the transformed task. The output generated by the execution is searched for a characteristic indicating the persistence of the bug, such as an assertion error or a segmentation fault. If the characteristic is found, the current version of the task is transformed again the same way. If the characteristic is not found, another task element is randomly chosen and removed from the task and tested for the characteristic. Our approach has similarities to *QuickCheck*, a tool for testing properties in Haskell programs [6]. *QuickCheck* generates random inputs for a program and checks whether a certain property holds for each input. If an input violates the property, it is shrunk with the objective to find a minimal failing input.

Our minimizer takes three arguments: (1) a command string for the planner execution causing the bug, (2) the characteristic, which is either a string to be looked for in the output, or a file containing the implementation of an output parser, and (3) the type of task element to be removed. For SAS^+ tasks, the current version accepts *operator* and *variable* as removable task element options. For PDDL tasks, it accepts *action*, *object* and *predicate*. Default deletion of a predicate symbol occurs by replacing literals containing the predicate with the *truth* value. This transformation is the equivalent to a projection [7] on all predicate symbols but the deleted one, since the parsed PDDL task is in negation normal form [10]. There is also the option of replacing each atom containing the predicate with truth or falsity. For the case in which we want to compare the output of one planner execution with the output of a second planner execution, a second command string and a second characteristic can be passed to the minimizer. When we use this option, each characteristic needs to be found in its respective planner execution for the search to continue.

Our minimizer is implemented in Python, the source code and a manual can be found at https://gitlab.com/galluc00/bachelor-thesis. Its design is strongly influenced by the Fast Downward Planning System [9], as some of its translator modules are re-used for PDDL parsing and task element

transformation. We perform all test runs during development and use-case analysis with Fast Downward. We demonstrate the functionality of the minimizer by testing it with three use-cases and different running configurations. In the most significant minimization result, we are able to reduce a SAS^+ task with 1536 operators and 184 variables to a task with 2 operators and 2 variables. After analyzing the resulting shrunk task with GDB [1], we believe it is still causing the same unexpected behavior as the original task.

In Chapter 2, we introduce definitions for the task types accepted by the minimizer and give a brief description of the optimization strategy. Chapter 3 describes how task transformations are done in detail and Chapter 4 reveals how our system is implemented. In Chapter 5, we describe the selected use-cases and show experimental results of the minimizer's performance. Finally, we reflect on the results and talk about potential extensions of our work in Chapter 6.

2
Background

In this chapter, we introduce relevant concepts for this thesis project. We provide a brief description of Automated Planning and a definition of the PDDL and FDR (SAS^+) tasks that are accepted by the Fast Downward Planning System [9], which the project is based upon. Furthermore, we describe the hill-climbing algorithm [11], which is used at the core of the minimizer.

2.1 Automated Planning

Automated Planning focuses on developing algorithms and heuristics for automatic solving of planning tasks. Planning tasks are state space search problems and can be modelled by means of an input language. The input language defines properties of the problem, such as an initial state, a goal and actions that lead from one state to another. The objective of planning is to find a valid action sequence, i.e., a *plan*, leading from the initial state to the goal.

2.2 PDDL and FDR (SAS^+) Tasks

The task minimizer implemented for this thesis uses Python modules that were developed for the translator of the Fast Downward Planning System. This means, the PDDL variant of our minimizer accepts the same kind of PDDL tasks Fast Downward does, which is "the *non-numerical*, *non-temporal* fragment of PDDL 2.2" [10]. In this section, we provide formal definitions for such a planning task. The following four definitions are from Helmert [2009] with a small modification: wherever the term *operator* was used in the context of PDDL in the original paper, we use the term *action*.

Definition 1 (PDDL actions)**.** *A **PDDL action** is a pair $\langle \chi, e \rangle$, which consists of a (possibly open) first-order formula χ called its **precondition** and a **PDDL effect** e. PDDL effects are recursively defined by finite application of the following rules:*

- *A first-order literal l is a PDDL effect called a **simple effect**.*
- *If $e_1, \ldots, e_n$ are PDDL effects, then $e_1 \wedge \cdots \wedge e_n$ is a PDDL effect called a **conjunctive effect**.*

- *If χ is a first-order formula and e is a PDDL effect, then $\chi \rhd e$ is a PDDL effect called a* ***conditional effect****.*
- *If $v_1, \ldots, v_k$ are variable symbols and e is a PDDL effect, then $\forall v_1 \ldots v_k : e$ is a PDDL effect called a* ***universally quantified effect*** *or* ***universal effect****.*

Free variables of simple effects are defined as for literals in first-order logic. Free variables of other effects are defined by structural induction:

- $\text{free}(e_1 \wedge \cdots \wedge e_n) = \text{free}(e_1) \cup \cdots \cup \text{free}(e_n)$
- $\text{free}(\chi \rhd e) = \text{free}(\chi) \cup \text{free}(e)$
- $\text{free}(\forall v_1 \ldots v_k : e) = \text{free}(e) \backslash \{v_1, \ldots, v_k\}$

The set of free variables of a PDDL action is defined as $\text{free}(\langle \chi, e \rangle) = \text{free}(\chi) \cup \text{free}(e)$*. Free variables are also called* ***parameters*** *of the action.*

Definition 2 (PDDL axioms)**.** *A* ***PDDL axiom*** *is a pair $\langle \varphi, \psi \rangle$ such that φ is a first-order atom and ψ is a first-order formula with* $\text{free}(\psi) \subseteq \text{free}(\varphi)$*. We write the axiom $\langle \varphi, \psi \rangle$ as $\varphi \leftarrow \psi$ and call φ the* ***head*** *and ψ the* ***body*** *of the axiom.*

A set $\mathcal{A}$ of PDDL axioms is called ***stratifiable*** *iff there exists a total preorder $\preceq$ on the predicate symbols of $\mathcal{A}$ such that for each axiom where predicate Q occurs in the head, we have $P \preceq Q$ for all predicates P occurring in the body, and $P \prec Q$ for all predicates P occurring in a negative literal in the translation of the body to negation normal form.*

Definition 3 (PDDL tasks)**.** *A* ***PDDL task*** *is given by a 4-tuple $\Pi = \langle \chi_0, \chi_\star, \mathcal{A}, O \rangle$ with the following components:*

- *χ_0 is a finite set of ground atoms called the* ***initial state****.*
- *$\chi_\star$ is a closed formula called the* ***goal formula****.*
- *$\mathcal{A}$ is a finite stratified set of PDDL axioms.*
- *O is a finite set of PDDL actions.*

Predicates occurring in the head of an axiom in $\mathcal{A}$ are called ***derived predicates****. Predicates occurring in the initial state or in simple effects of actions in O are called* ***fluent predicates****. The sets of derived and fluent predicates are required to be disjoint.*

Next, we provide the definition of FDR tasks. FDR tasks are an extension of the SAS$^+$ planning formalism [10]. They represent the type of task that is accepted by the search component of Fast Downward, and therefore one of the types the minimizer accepts for transformations. Later, we refer to FDR tasks simply by SAS$^+$ tasks.

Definition 4 (Planning tasks in finite-domain representation (FDR tasks))**.** *A* ***planning task in finite-domain representation (FDR task)*** *is given by a 5-tuple $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, O \rangle$ with the following components:*

- *$\mathcal{V}$ is a finite set of **state variables**, where each variable $v \in \mathcal{V}$ has an associated finite domain $\mathcal{D}_v$. State variables are partitioned into **fluents** (affected by operators) and **derived variables** (computed by evaluating axioms). The domains of derived variables must contain the **default value** $\bot$.*
  *A **partial variable assignment** over $\mathcal{V}$ is a function s on some subset of $\mathcal{V}$ such that $s(v) \in \mathcal{D}_v$ wherever $s(v)$ is defined. A partial variable assignment is called a **state** if it is defined for all fluents and none of the derived variables in $\mathcal{V}$. It is called an **extended state** if it is defined for all variables in $\mathcal{V}$. In the context of partial variable assignments, we write $v = d$ for the variable-value pairing $\langle v, d\rangle$ or $v \mapsto d$.*
- *s_0 is a state over $\mathcal{V}$ called the **initial state**.*
- *$s_\star$ is a partial variable assignment over $\mathcal{V}$ called the **goal**.*
- *$\mathcal{A}$ is a finite set of (FDR) **axioms** over $\mathcal{V}$. Axioms are triples $\langle cond, v, d\rangle$, where cond is a partial variable assignment called the **condition** or **body** of the axiom, v is a derived variable called the **affected variable**, and $d \in \mathcal{D}_v$ is called the **derived value** for v. The pair $\langle v, d\rangle$ is called the **head** of the axiom.*
  *The axiom set $\mathcal{A}$ is partitioned into a totally ordered set of **axiom layers** $\mathcal{A}_1 \prec \cdots \prec \mathcal{A}_k$ such that within the same layer, each affected variable must appear with a unique value in all axiom heads and bodies. In other words, within the same layer, axioms with the same affected variable but different derived values are forbidden, and if a variable appears in an axiom head, then it may not appear with a different value in a body. This is called the **layering property**.*
- *$\mathcal{O}$ is a finite set of (FDR) **operators** over $\mathcal{V}$. An operator $\langle pre, eff\rangle$ consists of a partial variable assignment pre over $\mathcal{V}$ called its **precondition**, and a finite set of **effects** eff. Effects are triples $\langle cond, v, d\rangle$, where cond is a (possibly empty) partial variable assignment called the **effect condition**, v is a fluent called the **affected variable**, and $d \in \mathcal{D}_v$ is called the **new value** for v.*

For axioms and effects, we commonly write $cond \rightarrow v := d$ in place of $\langle cond, v, d\rangle$.

2.3 Hill-Climbing

The goal of our implemented minimizer is to find a smaller task than the initial one that still fulfills some given property. We as users of the minimizer are mainly interested in the resulting task and not in the steps taken to reach it. When the path to a solution is irrelevant, local search algorithms can be a valid choice. In local search, only the current search node is stored and its neighborhood is considered in the search for better candidates. The simplicity and memory-efficiency of local search come at a price: (1) The algorithm might not find a solution, even though one exists, i.e., it is not *complete* and (2) a solution is found and the algorithm terminates, despite a better solution being available, i.e., it is not *optimal*.
Hill-climbing is considered the simplest variant of local search. In hill-climbing, the search process is initiated with an initial state. Its neighborhood is then evaluated by assigning each neighboring state a value, e.g., the estimated goal distance via a heuristic function. The most promising neighbor becomes the new search candidate and, again, its neighbors are considered. The algorithm terminates

when no neighbor's evaluation is better than the current candidate's. This may of course lead to local optima. Algorithm 1 describes hill-climbing in pseudo-code:

Algorithm 1 Hill-Climbing

```
1: current ← initial candidate
2: loop
3:     next ← a neighbor of current with maximal value
4:     if value of next ≤ value of current then
5:         return current
6:     end if
7:     current ← next
8: end loop
```

The hill-climbing algorithm we just described relies on a way to rank neighboring states by their values and chooses the most promising one. When the number of neighbors becomes very large, it might become infeasible to generate and store all of them, let alone rank them according to a heuristic or objective function. For those reasons, *first-choice hill-climbing* was used in the task minimizer implemented for this project. This variant of hill-climbing generates neighbors of the current task and picks the first higher-valued neighbor encountered as the new *current*. In Section 4.4 of Chapter 4, we describe how we use this algorithm.

3
Task Transformations

The implemented task minimizer offers three main options to shrink a given PDDL task: deleting actions, deleting predicate symbols and deleting objects. For SAS$^+$ tasks, the two implemented options are the deletion of operators and the deletion of variables. The deletion of actions in PDDL tasks is not very complex, as the action chosen to be deleted simply can be removed from the set of actions of the task. The same holds for operators in SAS$^+$ tasks. On the other hand, to correctly delete a predicate symbol, object or variable from a task, we need a more sophisticated approach. For example, if an object with the name "box" should be removed from a PDDL task, it is not sufficient to remove it from the set of objects of the task. If "box" still occurs in the initial state or the goal, this can lead to incorrect behavior of a planner. In the following sections, we define how the minimizer deletes different task elements from PDDL and SAS$^+$ tasks. It is important to note that task transformations performed by the minimizer do not preserve task semantics. The goal of task transformations in the scope of this thesis project is to obtain a smaller task through deletion of one of the described task elements.

3.1 Deletion of Actions

Since actions are not part of other task elements, their deletion is a simple process.

Definition 5 (Deletion of Actions)**.** *The deletion of action α from a given PDDL task entails the removal of α from the set of actions of the task.*

3.2 Deletion of Predicates

Wherever a predicate symbol P is a part of a greater formula, it does not make sense to simply delete it. In that case, the atom or literal containing P needs to be replaced with $\top$ (*truth*) or $\bot$ (*falsity*). By default, the minimizer replaces a literal containing P with $\top$. This corresponds to a projection [7] on all predicate symbols but P, since the parsed task is in negation normal form [10]. When running the minimizer, there are also the options of setting all atoms containing P to either $\top$ or $\bot$. After such a replacement, a method from the Fast Downward translator called `simplified` is called on the entire formula the replaced literal or atom was a part of, which gets rid of redundancies such as $\varphi \vee \top \equiv \top$ or $\varphi \wedge \top \equiv \varphi$ for formulas φ. The following definition is for the default case, where

literals containing the predicate symbol to be deleted are replaced with $\top$. The definitions for the two other options are very similar, except that, instead of literals being replaced with $\top$, atoms are replaced with $\top$ or $\bot$, respectively.

Definition 6 (Deletion of Predicate Symbols)**.** *The deletion of predicate symbol P from a given PDDL task entails the following operations:*

- *P is removed from the set of predicates of the task.*
- *Each atom in the initial state of the task containing P is removed from the initial state.*
- *Each literal containing P in the goal formula is replaced with* $\top$.
- *For each action, the following operations are performed:*
  - *Each literal containing P in the action precondition formula is replaced with* $\top$.
  - *If an action's precondition becomes* $\bot$*, the entire action is deleted.*
  - *Each literal containing P in the PDDL effect is replaced with* $\top$.
  - *If the PDDL effect is conditional, each literal containing P in the effect condition is replaced with* $\top$.
  - *If the PDDL effect is conditional and its condition becomes* $\bot$*, the entire conditional effect becomes* $\bot$.
  - *If the entire PDDL effect becomes* $\top$ *or* $\bot$*, the entire action is deleted.*
- *For each axiom, the following operations are performed:*
  - *If P occurs in the head of the axiom, the entire axiom is deleted.*
  - *Each literal containing P in the body of the axiom is replaced with* $\top$.
  - *If the body of the axiom becomes* $\bot$*, the entire axiom is deleted.*
  - *If the body of the axiom becomes* $\top$*, an artificial dummy predicate with arity 0 is created to become the new axiom body and act as a trigger for the axiom. An atom with this new predicate is also added to the initial state of the task.*

3.3 Deletion of Objects

The deletion of objects is simpler than the deletion of predicates in terms of complexity. Atoms in the goal formula containing the object to be deleted are replaced with $\top$.

Definition 7 (Deletion of Objects)**.** *The deletion of object* ω *from a given PDDL task entails the following operations:*

- ω *is removed from the set of objects of the task.*
- *Each atom in the initial state of the task containing* ω *is removed from the initial state.*
- *Each action containing* ω *is deleted.*
- *Each atom in the goal of the task that contains* ω *is replaced with* $\top$.

3.4 Deletion of Operators

The deletion of operators from SAS$^+$ tasks can be considered identical to the deletion of PDDL actions.

Definition 8 (Deletion of Operators)**.** *The deletion of operator o from a given SAS$^+$ task entails the removal of o from the set of operators of the task.*

3.5 Deletion of Variables

In the implementation, variables are identified via an index. Thus, in each transformation, when a variable with index i is deleted, we decrease the indices of all variables with index $> i$ in the entire task by 1.

Definition 9 (Deletion of Variables)**.** *The deletion of variable v from a given SAS$^+$ task entails the following operations:*

- *v is removed from the set of state variables of the task.*
- *For each mutex group, each fact containing v is removed.*
- *v is removed from the initial state.*
- *If a fact in the goal contains v, this fact is removed from the goal.*
- *For each operator, the following operations are performed:*
  - *If a fact in the prevail condition contains v, it is removed from the prevail condition.*
  - *If v is the affected variable in an effect, this effect is removed from the operator.*
  - *If a fact in an effect precondition contains v, it is removed from the effect precondition.*
  - *If no effect remains in the operator, the entire operator is deleted.*
- *For each axiom, the following operations are performed:*
  - *If v is the affected variable of the axiom, the axiom is deleted.*
  - *If a fact in the body of the axiom contains v, it is removed from the axiom body.*

4

Implementation

The implemented task minimizer is a command-line tool written in the Python programming language. Python is the language of choice because the translator component of the Fast Downward Planning System is also written in Python. Thus, we are able to re-use the code for PDDL parsing and the classes for PDDL and SAS$^+$ tasks. We also consider Python a powerful and flexible language; thus, personal preference undoubtedly influences the choice. The goal of this chapter is to explain the core functionality of the program and the algorithms used.

4.1 Overview

The prerequisite to using the minimizer (in a meaningful way) is a reproducible bug on the Fast Downward planner. The bug may be induced by the *translate*- as well as the *search*-component of the planner and PDDL and SAS$^+$ tasks are both allowed. A PDDL task must be present in two files, one for the domain description and one for the instance description.

The string replicating the planner execution causing the bug is passed on to the minimizer as argument. The minimizer performs transformations on the provided planning task in form of successive deletions of task elements of a specified kind, as described in Chapter 3. The type of task element to be successively removed is another argument that is required by the program. The current version of the minimizer supports the deletion of *actions*, *objects* and *predicates* for PDDL tasks and *operators* and *variables* for SAS$^+$ tasks. At each iteration of the task minimization, one randomly selected task element of the specified type is removed from everywhere it is used in the task and the planning problem is executed with the modified task, to check whether the bug is still produced. This check is done by testing for the presence of a characteristic string in the command-line output generated by the execution of the planner. Thus, this string is the third necessary argument needed for an execution of the minimizer. Instead of the characteristic string, there is also the possibility to provide the path to the implementation of a simple parser interface, `parserbase.py`. The `ParserBase` interface has one method to be implemented, namely `parse_output_string`, which takes the output string as parameter and returns a boolean. The user of the program has more flexibility in how to process the output of the planner execution when using the parser. Another option for the user is to provide a second execution command and a second characteristic string or parser implementation. The condition for continuation of the search is then that each planner execution must reproduce the

respective characteristic. The program terminates the deletion of task elements of a specific type, when none of the remaining possible deletions leads to the characteristic(s) being reproduced.

4.2 User Interface

A correct execution call of the minimizer has the following structure:

```
$ python3 minimizer.py --problem PROBLEM [PROBLEM]
                       --characteristic CHARACTERISTIC [CHARACTERISTIC]
                       --delete OPTION [OPTION ...]
                       [--truth]
                       [--falsity]
                       [--write-summary]
```

- `PROBLEM`: The command string that replicates the bug on the planning system. A second `PROBLEM` argument is interpreted as a reference planner execution command and requires a second `CHARACTERISTIC` argument.
- `CHARACTERISTIC`: A characteristic string that should appear in the output after the execution of `PROBLEM` or, alternatively, the path to a Python file implementing the provided parser interface `parserbase.py`. A second `CHARACTERISTIC` argument is interpreted as the characteristic for the execution of the reference planner.
- `OPTION`: The specification of the task element type to be deleted in the planning task. `OPTION` $\in$ {*action*, *object*, *predicate*} for PDDL tasks and `OPTION` $\in$ {*operator*, *variable*} for SAS$^+$ tasks.

4.3 Internal Problem Representation

At the beginning of the program execution, the program arguments are either passed to class `PDDLMinimizationProblem` or `SASMinimizationProblem` to create an instance of the minimization problem. These classes handle the execution and minimization of the problem and the testing for the presence of the characteristic string. After the instantiation of the problem, the main script `minimizer.py` calls the method `minimize()` on the minimization problem, which invokes the minimization process on the planning task and returns the processed task. The duration of the minimization process depends on the size and complexity of the planning task. The program terminates by writing the processed planning task into a single file `minimized.sas` for SAS$^+$ tasks or into `minimized-domain.pddl` and `minimized-problem.pddl` for PDDL tasks. A simplified overview of the classes handling the minimization problems is provided in Figure 4.1.

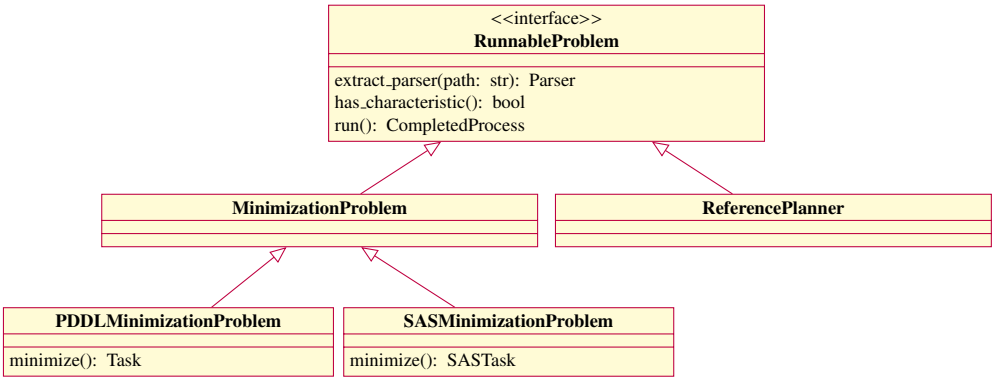

Figure 4.1: Simplified overview of the minimization problem classes.

4.4 Task Transformations

The `minimize()` method will sequentially repeat the minimization process for each `OPTION` of *delete* provided with the parameters. As mentioned in Section 2.3, first-choice hill-climbing [11] was used as optimization algorithm. The following pseudo-code describes the algorithm:

Algorithm 2 First-Choice Hill-Climbing

```
1:  current ← initial candidate
2:  loop
3:      for succ ∈ successors of current do
4:          if succ has characteristic then
5:              current ← succ
6:              break
7:          end if
8:      end for
9:      if none of the successors had the characteristic then
10:         return current
11:     end if
12: end loop
```

First-choice hill-climbing (Algorithm 2) generates one successor at a time in random order by deleting one task element of the specified type (line 3). As soon as a successor replicates the bug, it becomes the new *current* and its successors are considered, deepening the search level (lines 5 and 6, followed by a restart at line 3). If the for-loop makes it through all successors without any of them replicating the bug (line 9), then the last one to do so is returned and the search terminates (line 10). The reason for the choice of this variant of hill-climbing was the lack of a practical heuristic to (efficiently) distinguish the quality of successor tasks. The design and implementation of such a heuristic could be a meaningful extension to the minimizer, as it may speed up the minimization process or improve the quality of the results.

4.4.1 Transformation of PDDL Tasks

The transformation of PDDL tasks was implemented using the *visitor pattern* [8]. This programming pattern allows adding functionality to a class externally, without adding much code to the class itself. Instead, the class to be extended implements simply one additional method called `accept`, which takes a visitor object as argument. One separate visitor class is implemented for each type of task transformation. Visitor classes have methods with each the responsibility of transforming one specific task element (actions, conditions, ...) or task element parts (action effects, condition subtypes, ...). The `accept` method of a task element is only responsible for calling the correct `visit` method of the received visitor with the task element itself as argument. Nested expressions, e.g., in action or effect conditions, make use of recursive calls of `visit` methods in order to perform the correct transformations at the lowest level (literals). With this pattern, the modification of task elements is fully implemented in the visitor classes, which keeps the code clean. Figure 4.2 illustrates the implemented visitor classes and shows their dependencies. It is worth mentioning that these dependencies are neither strictly of the nature *inherits-from* nor *implements*, because in Python these constructs are not automatically enforced. Rather, each visitor class implements the functions that are needed for its transformations and implementing the visitors in this hierarchy allowed for a more rigorous engineering process.

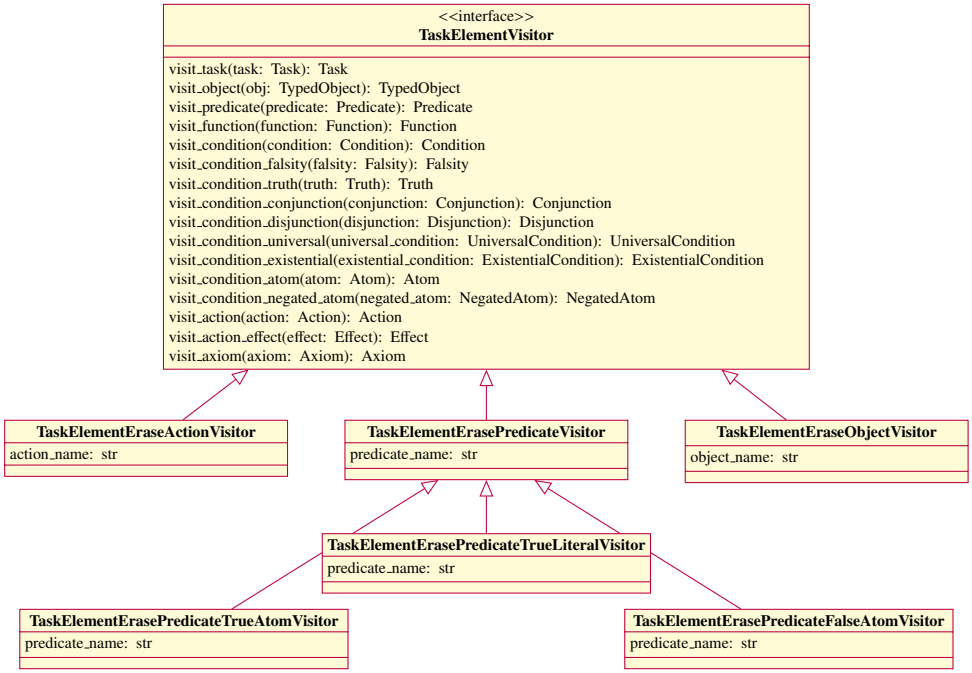

Figure 4.2: Simplified visualization of the dependencies between visitor classes.

Task transformations by visitors are invoked by the `get_successors` method of one of the subclasses of the `Transformer` interface. The only thing `get_successors` does, is randomly choosing the next candidate to be deleted and calling the task's `accept` method with a visitor initialized with the name of the candidate. Figure 4.3 illustrates the transformer classes and their dependencies.

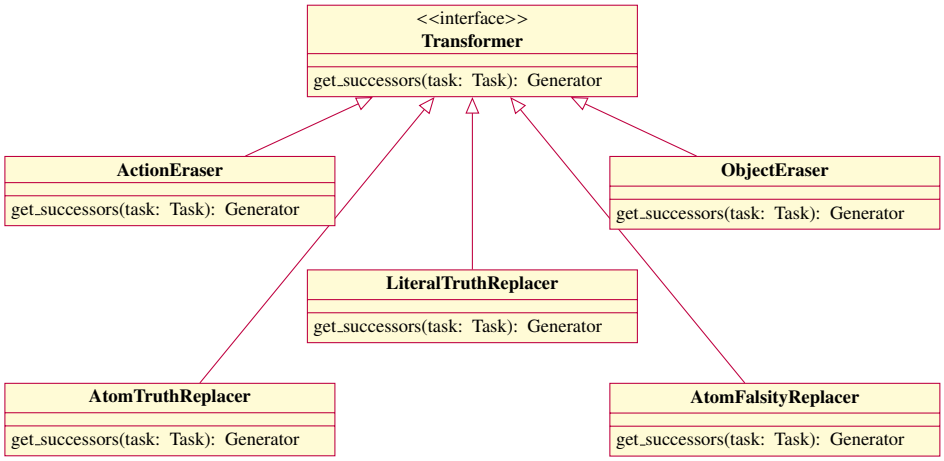

Figure 4.3: Overview of PDDL transformer classes.

4.4.2 Transformation of SAS$^+$ Tasks

Since SAS$^+$ tasks have a much simpler structure than PDDL tasks, there is no need for a sophisticated transformation method such as the visitor pattern mentioned in Section 4.4.1. Instead, transformations are directly performed from outside the `SASTask` objects by a single `transform` method implemented for each subtype of the `SASTransformer` interface. Deleting an operator is as trivial as removing operator o from the list of operators and otherwise leave the task unchanged. Deleting a variable from a SAS$^+$ task takes more effort, as the variable must be removed from all occurrences in mutexes, the initial state, the goal, operators and axioms. Additionally, if variable with index i is removed from the task, in all occurrences of variables with indices $> i$, a decrement of their indices by 1 must occur for the transformation to be correct. Figure 4.4 illustrates the transformers implemented for SAS$^+$ task transformation.

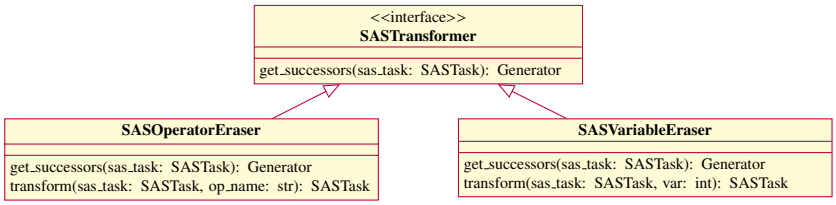

Figure 4.4: Overview of SAS$^+$ transformer classes.

4.5 Other Modules

There are two additional modules of the implementation worth mentioning, namely:

- The `pddl_writer` module, which contains procedures for writing a PDDL task into a domain and a problem file.
- The `sas_reader` module, which contains functions for parsing a SAS$^+$ file in order to obtain an instance of `SASTask`.

5 Use-Cases

The functionality of the implemented task minimizer is illustrated in this chapter, based on three use-cases. These use-cases are a selection from bug-reports of Fast Downward users and from the planner's official list of issues[1]. The fact that these bugs are considered resolved at the time of writing of this report is irrelevant, since the only goal is to reproduce them on the respective version of the planner. We executed the test runs of the minimizer on a laptop computer with the following specifications:

- Operating system: Ubuntu 20.04
- Processor: Intel i5-5300U, 2 cores, 2.3 GHz base frequency
- Memory: 11.4 GiB

We believe memory bottlenecks occurred in some runs, which would explain longer running times, e.g., with delete option *predicate (truth)* in Table 5.1. We chose to ignore predicate deletion options *truth* and *falsity* in test runs where all three PDDL transformation options were used, to keep the number of test runs reasonable. The averages and standard deviations are computed from 5 runs of each configuration. Since we observed a high volatility in running times even with a fixed randomization seed, we used the fixed seed 42 for all test runs. For a cleaner appearance, throughout this entire chapter, full file paths (or parts of them) are aliased with `/path/to/`.

5.1 Use-Case 1: Issue 335

This issue was used to develop and test the PDDL version of the task minimizer, despite it being the most outdated one. It has been reported in May 2012 and was marked as resolved in August 2014 [3]. The planning task triggering the bug is described in the files `cntr-domain.pddl` and `cntr-problem.pddl`. The task requires the ADL fragment of PDDL and the initially parsed task instance features 46 actions, 62 objects and 29 predicate symbols. On the revision of Fast Downward with commit hash `09ccef5fd[...]` (from 04/08/2014), which is the parent com-

[1] http://issues.fast-downward.org/

mit of the one where the bug was fixed, the following error can be reproduced when executing `src/translate/translate.py` with the two above mentioned files:

```
Traceback (most recent call last):
  File "translate/translate.py", line 677, in <module>
    main()
  File "translate/translate.py", line 667, in main
    sas_task = pddl_to_sas(task)
  File "translate/translate.py", line 540, in pddl_to_sas
    implied_facts)
  File "translate/translate.py", line 422, in translate_task
    actions, axioms, goals)
  File "/path/to/downward/src/translate/axiom_rules.py", line 11, in handle_axioms
    axioms = compute_negative_axioms(axioms_by_atom, axiom_literals)
  File "/path/to/downward/src/translate/axiom_rules.py", line 158, in compute_negative_axioms
    new_axioms += negate(axioms_by_atom[literal.positive()])
  File "/path/to/downward/src/translate/axiom_rules.py", line 168, in negate
    assert len(condition) > 0, "Negated axiom impossible; cannot deal with that"
AssertionError: Negated axiom impossible; cannot deal with that
```

The last line of the error message (*AssertionError: Negated axiom impossible; cannot deal with that*) can be used as the characteristic string for the minimizer. In Table 5.1, we illustrate the average running times and the resulting minimized task elements for a set of configurations. The minimizer command we use for the tests (with the respective delete option) is the following:

```
python3 /path/to/minimizer.py --problem "python3.7 /path/to/downward/src/translate/translate.py /path/to/
     cntr-domain.pddl /path/to/cntr-problem.pddl" --characteristic "AssertionError: Negated axiom
     impossible; cannot deal with that" --delete predicate object action
```

delete option(s)	total time	time per option: action	time per option: object	time per option: predicate	before → after: actions	before → after: objects	before → after: predicates
action	251.087 ± 6.869s	251.087 ± 6.870s			46 → 4	62 → 62	29 → 29
object	83.424 ± 1.422s		83.424 ± 1.422s		46 → 10	62 → 8	29 → 29
predicate	427.928 ± 123.118s			427.928 ± 123.118s	46 → 41	62 → 62	29 → 9
predicate (truth)	1594.830 ± 233.849s			1594.830 ± 233.849s	46 → 5	62 → 62	29 → 6
predicate (falsity)	124.378 ± 12.732s			124.377 ± 12.731s	46 → 11	62 → 62	29 → 9
action, object, predicate	333.605 ± 18.559s	246.430 ± 14.675s	84.850 ± 3.892s	2.325 ± 0.127s	46 → 4	62 → 13	29 → 4
action, predicate, object	313.533 ± 7.165s	263.337 ± 10.606s	7.118 ± 0.798s	43.077 ± 4.134s	46 → 2	62 → 6	29 → 4
object, action, predicate	89.107 ± 3.503s	0.819 ± 0.074s	85.816 ± 3.349s	2.472 ± 0.389s	46 → 2	62 → 8	29 → 4
object, predicate, action	94.215 ± 11.503s	0.531 ± 0.210s	90.811 ± 10.593s	2.872 ± 0.754s	46 → 2	62 → 8	29 → 5
predicate, action, object	552.037 ± 46.078s	324.719 ± 28.436s	8.549 ± 0.600s	218.769 ± 18.650s	46 → 2	62 → 7	29 → 9
predicate, object, action	255.320 ± 25.537s	0.372 ± 0.028s	34.736 ± 0.914s	220.212 ± 24.814s	46 → 2	62 → 7	29 → 9

Table 5.1: Average results from 5 runs of each selected configuration of the minimizer with the task from issue 335.

Between runs with the same delete option configuration, we did not observe different minimization outcomes. The results do not make it obvious, which order of delete options is ideal. If we define the sum of the remaining number of task elements as a metric for the minimization success, e.g., 21 for *action, object, predicate*, we can discuss the tradeoff between the total minimization duration and how minimized the task becomes. For this use-case, delete option configuration *action, predicate, object* yields the best overall minimization result (12, according to the just defined metric) at an average running time of ~314s. However, configuration *object, action, predicate* yields a slightly worse minimization result (14), but at less than a third of the time, ~89s on average. At this point, it is up to the user of the minimizer to decide which of these properties is more important. Overall, it is pleasing to see how much the size of the task could be reduced by the minimizer, while still triggering the same bug.

5.2 Use-Case 2: *Segmentation Fault with operator-counting heuristic and state equation contraints*

In August 2020, a bug report with the above title was issued in the Fast Downward public mailing list [2]. The author implemented a Petri Net translation from PDDL and obtained a segmentation fault from Fast Downward's search component when using the following two heuristics options with the A* algorithm:

- operatorcounting(constraint_generators=[state_equation_constraints()])
- operatorcounting(constraint_generators=[state_equation_constraints(),lmcut_constraints()])

In this use-case analysis, we only consider the first of the two heuristics options to reproduce the bug and test the minimizer. With the provided SAS$^+$ file `output_petri_sokobanp01.sas`, we are able to reproduce the bug. The initially parsed SAS$^+$ task instance contains 1536 operators and 184 variables. Since this bug was issued on 01/08/2020, the most recent Fast Downward release (20.06) can be used to reproduce it. Specifically, we are using the Git tag `release-20.06.0` with commit hash `3a27ea77f[...]` from 26/07/2020 for the experiments. In order to run this use-case's configuration, an LP solver is required. We use IBM's CPLEX for all test runs and experiments.
Executing the Fast Downward search component with the problem from above yields the following output:

```
[t=0.000440241s, 45108 KB] reading input...
[t=0.0233119s, 46304 KB] done reading input!
[t=0.0252104s, 47280 KB] Initializing constraints from state equation.
[t=0.0392132s, 51380 KB] Building successor generator...done!
[t=0.0426499s, 51380 KB] peak memory difference for successor generator creation: 0 KB
[t=0.0426796s, 51380 KB] time for successor generation creation: 0.00317523s
[t=0.0427079s, 51380 KB] Variables: 184
[t=0.0427337s, 51380 KB] FactPairs: 368
[t=0.0427552s, 51380 KB] Bytes per state: 24
[t=0.0428768s, 51380 KB] Conducting best first search with reopening closed nodes, (real) bound = 2147483647
[t=0.0543639s, 53500 KB] New best heuristic value for operatorcounting(constraint_generators = list(
     state_equation_constraints)): 1
[t=0.0544395s, 53500 KB] g=0, 1 evaluated, 0 expanded
[t=0.0544747s, 53500 KB] f = 1, 1 evaluated, 0 expanded
[t=0.0545164s, 53500 KB] Initial heuristic value for operatorcounting(constraint_generators = list(
     state_equation_constraints)): 1
[t=0.0545561s, 53500 KB] pruning method: none
Peak memory: 53500 KB
caught signal 11 -- exiting
Segmentation fault (core dumped)
```

Again, we can use the last line of the error output (*Segmentation fault (core dumped)*) as characteristic string for the minimizer.
The following command is used for our experiment runs:

```
python3 /path/to/minimizer.py --problem "/path/to/downward/builds/release/bin/downward --search 'astar(
     operatorcounting(constraint_generators=[state_equation_constraints()]))' --internal-plan-file sas_plan
      < /path/to/output_petri_sokobanp01.sas" --characteristic "Segmentation fault (core dumped)" --delete
     variable operator
```

It turns out that after a few runs of this configuration and *operator* as delete option, we are able to observe what appear to be non-deterministic results for the minimized task: after some runs, there are 18 operators remaining and after some, there are 35. After narrowing down where these two types of runs start to differ, we are able to observe the following error output:

```
[t=0.000508709s, 45112 KB] reading input...
[t=0.0125704s, 45788 KB] done reading input!
[t=0.0144455s, 46848 KB] Initializing constraints from state equation.
[t=0.0222302s, 49184 KB] Building successor generator...done!
[t=0.0237845s, 49184 KB] peak memory difference for successor generator creation: 0 KB
[t=0.0238248s, 49184 KB] time for successor generation creation: 0.00135768s
[t=0.0238734s, 49184 KB] Variables: 184
[t=0.0238988s, 49184 KB] FactPairs: 368
[t=0.0239504s, 49184 KB] Bytes per state: 24
[t=0.0240734s, 49184 KB] Conducting best first search with reopening closed nodes, (real) bound = 2147483647
realloc(): invalid pointer
Peak memory: 50364 KB
caught signal 6 -- exiting
Aborted (core dumped)
```

We believe this behavior is caused by a race condition, in which one of the signals comes from the LP solver and the other one from the system and the one reaching the planner first causes the error. This would explain the non-deterministic nature of the result. However, we do not investigate this issue any further for this project. Instead, we treat these varying minimization results like the recorded running times and take their averages and standard deviations, as can be seen in Table 5.2. While all four running configurations in Table 5.2 show pleasing results in the minimization of the task elements, configuration *variable, operator* stands out, reducing the initial SAS$^+$ task to only 2 variables and 2 operators. To investigate whether the minimized task with 2 variables and 2 operators still triggers the same bug as the original task, we can run the particular Fast Downward version with GDB, The GNU Project Debugger [1], separately, and compare the outputs. The fact that these outputs are nearly identical gives us reason to believe that the same bug is still triggered with the minimized task.

Another possibility would be to treat both error outputs as qualifying characteristic strings, which can be achieved implementing the parser interface the following way, for example:

```
from parserbase import ParserBase

class Parser(ParserBase):
    def parse_output_string(self, output_string) -> bool:
        cond1 = "caught signal 11 -- exiting" in output_string
        cond2 = "caught signal 6 -- exiting" in output_string
        return cond1 or cond2
```

That way, one would likely always obtain the same number of remaining operators after minimization.

		time per option		before → after	
delete option(s)	total time	operator	variable	operators	variables
operator	478.437 ± 13.088s	478.437 ± 13.088s		1536 → 24.8 ± 9.3	184 → 184
variable	58.908 ± 2.208s		58.908 ± 2.209s	1536 → 432	184 → 2
operator, variable	487.156 ± 25.638s	451.564 ± 22.753s	35.591 ± 3.306s	1536 → 25.6 ± 10.4	184 → 12.2 ± 6.9
variable, operator	138.751 ± 4.799s	78.822 ± 1.286s	59.928 ± 5.866s	1536 → 2	184 → 2

Table 5.2: Average results from 5 runs of each configuration of the minimizer with the task from the Segmentation Fault bug report.

5.3 Use-Case 3: Issue 736

This use-case from September 2017 is another issue borrowed from the official Fast Downward issue list [4]. Due to a bug in the translator, the attached planning task with files `domain.pddl` and `problem.pddl` is claimed to be unsolvable, despite there being an existing solution. For this use-case, we make use of the available option of passing a second planner to the minimizer with its separate characteristic. The idea is to pass a recent version of Fast Downward (commit `e9c2370e6[...]` from 27/07/2020) as reference planner, since it finds a plan for the task. Each transformed task instance then has to remain unsolvable with the first planner while at the same time remain solvable with the second one. The characteristic strings we want to look for in transformed task instances can be set to "*Search stopped without finding a solution.*" for the bug inducing version of the planner and "*Solution found.*" for the recent one.
We use the following command for our experiment runs:

```
python3 /path/to/minimizer.py --problem "/path/to/downward/./fast-downward.py /path/to/domain736.pddl /path/
    to/problem736.pddl --search 'astar(blind())'" "/path/to/downward_20.06/./fast-downward.py /path/to/
    domain736.pddl /path/to/problem736.pddl --search 'astar(blind())'" --characteristic "Search stopped
    without finding a solution." "Solution found." --delete predicate object action
```

For the more common case, where no working version of the same planner is available, any other planner, for which is known that it solves the task, could be used as the reference planner. We only tested the minimizer with Fast Downward. Table 5.3 illustrates the running times and minimization results for a set of running configurations and the two-planner setup described above.

		time per option			before → after		
delete option(s)	total time	action	object	predicate	actions	objects	predicates
action	0.836 ± 0.012s	0.836 ± 0.012s			3 → 3	5 → 5	6 → 6
object	1.492 ± 0.064s		1.492 ± 0.064s		3 → 3	5 → 5	6 → 6
predicate	2.753 ± 0.066s			2.753 ± 0.066s	3 → 3	5 → 5	6 → 3
predicate (truth)	2.734 ± 0.025s			2.734 ± 0.025s	3 → 3	5 → 5	6 → 3
predicate (falsity)	1.807 ± 0.027s			1.807 ± 0.028s	3 → 3	5 → 5	6 → 6
action, object, predicate	5.031 ± 0.088s	0.868 ± 0.012s	1.425 ± 0.029s	2.737 ± 0.065s	3 → 3	5 → 5	6 → 3
action, predicate, object	5.461 ± 0.119s	0.854 ± 0.042s	1.933 ± 0.036s	2.673 ± 0.060s	3 → 3	5 → 3	6 → 3
object, action, predicate	5.123 ± 0.235s	0.929 ± 0.165s	1.462 ± 0.066s	2.731 ± 0.062s	3 → 3	5 → 5	6 → 3
object, predicate, action	5.181 ± 0.170s	1.030 ± 0.088s	1.445 ± 0.025s	2.705 ± 0.085s	3 → 1	5 → 5	6 → 3
predicate, action, object	5.648 ± 0.119s	1.016 ± 0.020s	1.900 ± 0.039s	2.732 ± 0.071s	3 → 1	5 → 3	6 → 3
predicate, object, action	5.434 ± 0.106s	0.933 ± 0.034s	1.874 ± 0.020s	2.627 ± 0.078s	3 → 1	5 → 3	6 → 3

Table 5.3: Average results from 5 runs of each selected configuration of the minimizer with the task from issue 736.

Between runs with the same delete option configuration, we did not observe different minimization outcomes. When comparing the configurations with all three delete options, it becomes evident that the order of the options determines the level of minimization. If we again evaluate the minimization outcomes by the sum of remaining actions, objects and predicates, the two runs with *predicate* as first argument become the winning ones. The *predicate, object, action* run becomes the overall winner, due to a slightly shorter average runtime between the two.

6
Conclusions

The goal of this thesis project was to implement a planning task minimizer for finding small instances of PDDL or SAS$^+$ tasks causing some unexpected behavior on a planning system. The search for smaller task instances in the implemented minimizer is realized by successive random selection and removal of a task element of one of a few specific types, which are *actions*, *objects* and *predicates* for PDDL tasks and *operators* and *variables* for SAS$^+$ tasks. Task elements are removed in a syntactically consistent way from every occurrence in the task. However, no semantic integrity of tasks is guaranteed after this minimization process.

6.1 Evaluation of Use-Cases

We illustrate the functionality of the minimizer in Chapter 5 on the basis of three use-cases featuring planning tasks causing incorrect or unexpected results on the Fast Downward Planning System. In Section 5.1, we introduce the first tested use-case, which initially is made up of, among others, 46 actions, 62 objects and 29 predicate symbols. According to the measurements in Table 5.1, the overall best result in terms of the number of remaining task elements is achieved with delete option order *action, predicate, object*, shrinking the task to 2 actions, 6 objects and 4 predicates. In use-case 2, we run the minimizer on a SAS$^+$ planning problem initially containing, among others, 1536 operators and 184 variables. From the results presented in Table 5.2, we can see that, in terms of the greatest reduction of task elements, delete option order *variable, operator* does the best job. Runs with this configuration reduce the amount of operators as well as the amount of variables to 2 in the remaining task. In use-case 3, the best runs (by the same metric as above) are the two whose delete options start with *predicate*, followed by *object* and *action* in arbitrary order (Table 5.3). In these runs, the PDDL task initially containing, among others, 3 actions, 5 objects and 6 predicates is reduced to 1 action, 3 objects and 3 predicates. There are two additional remarks we want to make regarding the results of the use-cases:

1. The current version of the minimizer is not able to determine an optimal order of delete options by itself. The *best order* of options, as we called it above, solely represents the apparent best order for a specific use-case, compared to test runs with different configurations. Not all possible configurations were tested.

2. We choose not to compare the average runtimes of different runs in this thesis, because we want to keep the focus on the minimization results. We do believe there are improvements that could be made to the minimizer that may decrease search time or improve task minimization or both. We discuss ideas for future work in Section 6.2.

6.2 Future Work

There are multiple ways the task minimizer could be extended to improve its functionality. Currently, the search is rather uninformed, as it uses first-choice hill-climbing. Some kind of meta-heuristic could be used to estimate the size of successor tasks, so the smallest one could be picked each iteration. An idea for this meta-heuristic would be to count, for example, which object appears most times in the current task. This object would be the next one to be removed from the task by the transformer. Such a meta-heuristic could also eliminate the need for the user to specify which task element should be deleted – the minimizer could choose the best option each iteration by itself. Implementing additional delete options like axioms or mutexes or finding other ways to parameterize the search space could also be useful extensions. Currently, planner runs have to terminate before the output can be parsed. This is a waste of time if the user is interested in some planner output that is available early on. In the scope of this thesis, we have not investigated whether there are ways to parse generated output on the fly. If this can be achieved, the execution could be terminated prematurely, as soon as the characteristic is found. Finally, re-implementing the minimizer in a compiled language such as C++ could improve performance significantly as well.

Bibliography

[1] GDB : The GNU Project Debugger. https://www.gnu.org/software/gdb/ (accessed 01 October 2020).

[2] Segmentation fault with operatorcounting heuristic and state_equation_contraints. https://groups.google.com/g/fast-downward/c/34Y-jgYMZvc/m/91SJkQ4AAQAJ, 1 August 2020 (accessed 28 September 2020).

[3] Translate fails with "negated axiom" error. http://issues.fast-downward.org/issue335, 14 May 2012 (accessed 28 September 2020).

[4] Simplification of conditional effects is too agressive. http://issues.fast-downward.org/issue736, 15 September 2017 (accessed 30 September 2020).

[5] Carlos Areces, Facundo Bustos, Martín Ariel Domínguez, and Jörg Hoffmann. Optimizing planning domains by automatic action schema splitting. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 2014.

[6] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4):53–64, 2011. doi: 10.1145/1988042.1988046.

[7] Stefan Edelkamp. Planning with pattern databases. In *Proc. ECP*, volume 1, pages 13–24, 2001.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of reusable object-oriented sofware*. Addison Wesley, 2009.

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

[10] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[11] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2010.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Lucas Galery Käser

Matriculation number — Matrikelnummer
15-051-618

Title of work — Titel der Arbeit
Finding Small Counterexamples of Expected Planner Behavior with Hill-Climbing

Type of work — Typ der Arbeit
Bachelor thesis

Declaration — Erklärung
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 7 October 2020

Signature — Unterschrift