Optimality Certificates for Classical Planning

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, Remo Christen

Esther Mugdan
esther.mugdan@unibas.ch
2019-053-024

06.06.2022

Abstract

In general, it is important to verify software as it is prone to error. This also holds for solving tasks in classical planning. So far, plans in general as well as the fact that there is no plan for a given planning task can be proven and independently verified. However, no such proof for the optimality of a solution of a task exists. Our aim is to introduce two methods with which optimality can be proven and independently verified. We first reduce unit cost tasks to unsolvable tasks, which enables us to make use of the already existing certificates for unsolvability. In a second approach, we propose a proof system for optimality, which enables us to infer that the determined cost of a task is optimal. This permits the direct generation of optimality certificates.

Table of Contents

Abstract **ii**

1 Introduction **1**

2 Background **3**
2.1 Planning Tasks . . . 3
2.1.1 Cost . . . 4
2.2 PDDL . . . 4
2.3 Search Algorithms . . . 6
2.3.1 Blind Search . . . 6
2.3.2 A^* Search . . . 7
2.4 Certificates . . . 7

3 Reduction to Unsolvability **9**
3.1 Domain . . . 9
3.2 Task . . . 10
3.3 Individual Cost . . . 11

4 Experiments **13**
4.1 Results . . . 13
4.2 Conclusion . . . 18

5 Proof System for Optimality **19**
5.1 General Concept . . . 19
5.2 Formal Proof System . . . 20
5.2.1 Derivation Rules . . . 20
5.2.2 Basic Statements . . . 22

6 Certificates for Optimality **24**
6.1 Blind Search . . . 25
6.1.1 Proof of Premises for Blind Search . . . 25
6.1.2 Proof Sketch for Blind Search . . . 26
6.2 A^* Search . . . 26
6.2.1 Proof of Premises for A^* Search . . . 27
6.2.2 Proof Sketch for A^* Search . . . 28

6.3 h^{max} Heuristic 29
6.3.1 Proof of h-states for h^{max} 30

7 Conclusion **32**

Bibliography **34**

Appendix A Figures **35**

Declaration on Scientific Integrity **37**

1 Introduction

A classical planning task consists of a domain, which defines the universe of the task, and the task itself, which is given by an initial configuration as well as a goal. In each state of a task, we can apply actions which lead to a new state. In this way, we can traverse the states of our task. Each such action has a dedicated cost. When applying multiple actions one after the other, we sum up the cost over the individual action costs.

The solution of a task is a sequence of actions which can be performed to reach some goal configuration from the start configuration. This sequence of actions has to be applied consecutively to the states of the task. The first action leads from the initial state to some successor state, and the last action has a goal state as its successor.

A planning system can compute such a sequence of actions, which can either be a solution independent of the cost or a so-called optimal solution where the sum of the action costs is minimal. If no sequence can be found, the system will state that the problem is unsolvable. This outcome should ideally be independently verified, since planning systems can contain bugs and do not always provide a correct output.

Validating a sequence of actions for a solvable task is fairly simple. It just requires the verification that starting in the initial state and applying the actions consecutively will lead to a goal state. This can be performed with the help of validation tools already available.

However, verifying that a problem is unsolvable is harder, since we cannot simply check the output sequence. Instead, a certificate is created alongside the output. It states that the output, in this case the statement that the task is unsolvable, is correct. Such a certificate must again be validated by a verifier. The generation of unsolvability certificates which can be verified efficiently was first developed with the help of inductive sets [2] and further refined in an approach using a rule-based proof system [3].

Currently, it is not possible to verify that a plan is optimal. In this thesis, we aim to close this gap by creating certificates for optimal solutions, making it possible to verify that the solution provided by a planning system is a solution with the lowest cost for the task.

A first approach is to reduce a task with a path of optimal cost x to an unsolvable task. If the computed optimal solution has cost x, it follows that there cannot exist a solution with costs smaller than x. By reformulating our initial problem, stating that a solution is now allowed to have costs of at most $x - 1$, we should get an unsolvable task. Since there exist certificates for unsolvable tasks, we can use this method to validate that a solution is optimal.

Since we have to first solve the original task, then reformulate it and finally solve the reformulated task, the first approach is very time-consuming. In addition, bugs may occur during reformulation. Lastly, since we modify the original task and solve it again, this approach does not reflect the reasoning used by the algorithm to prove why the solution is optimal.

Therefore, we present a second approach, which aims to create optimality certificates by employing similar methods as are used for unsolvability certificates. In this case, we would not have to reformulate the problem but can use the given task itself and prove that there does not exist a cheaper solution.

2
Background

In this thesis, we consider planning tasks using the STRIPS representation [4]. We will also introduce PDDL, a language to represent such a planning task. In order to solve a task, we can make use of different algorithms, two of which we will present in the following. Lastly, we will introduce certificates for unsolvable tasks which make use of the algorithms mentioned. These definitions are needed in order to be able to derive optimality certificates later on.

2.1 Planning Tasks

We define a STRIPS planning task as a tuple $\Pi = \langle V, A, I, G \rangle$, where V is the finite set of state variables. These state variables are binary, i.e., they can either be true or false. A is the set of actions which can be performed in the various states. The initial state is denoted by I and the goal of the task by G.

A state s is defined by the state variables which are true in s, hence $s \subseteq V$. We can therefore write a state s as the set $s = \{v \mid v \in V, v = \mathit{true}\}$. The initial state I is defined in the same way. A state s is a goal state if all state variables which are required to be true in the goal are true in s, hence $G \subseteq s$. We call the set of all goal states S_G.

Each action a is defined by its preconditions $\mathit{pre}(a) \subseteq V$ and its add and delete effects $\mathit{add}(a) \subseteq V$ and $\mathit{del}(a) \subseteq V$. An action a can be applied in a state s if it fulfils its preconditions, i.e. $\mathit{pre}(a) \subseteq s$. Applying an action a in a state s results in a successor state $s[a]$. This state consists of the variables in s minus the variables which were removed by a delete effect and plus the variables which were added by an add effect. Therefore, $s[a] = (s \setminus \mathit{del}(a)) \cup \mathit{add}(a)$ must hold. For a set of states S and a set of actions $A' \subseteq A$, we write the set of all successor states of S as $S[A'] = \{s[a] \mid s \in S \text{ and } a \in A' \text{ applicable in } s\}$. Similarly, we formulate the set of all predecessors of S, namely $[A']S = \{s \mid s[a] \in S \text{ and } a \in A' \text{ applicable in } s\}$.

We say that a certain state s_n is reachable from another state s_0 if there exists a sequence of actions $\pi = \langle a_0, \ldots, a_{n-1} \rangle$ with a_i applicable in s_i and $s_i[a_i] = s_{i+1}$ for all $0 \leq i \leq n-1$. We can apply the same concept to sets of states. A state s_n is reachable from a set of states S if there exists a state $s \in S$ from which s_n is reachable.

A plan is a sequence of actions with which a goal state can be reached from another state. The sequence is started in a state s and has a goal state as the last successor state. Such a plan is called *s-plan*. In order to solve a task, we start in the initial state I. We therefore call the solution to a *solvable* task *I-plan*. If there exists no such plan, the task is *unsolvable*.

2.1.1 Cost

The above definition of a planning task is sufficient to determine if a task is solvable or unsolvable. Here, we want to discuss optimal plans for a task. An optimal solution for a task is a solution with the lowest possible cost.

Costs are introduced as an attribute c_a to each action a. This cost can be positive or zero. The cost of a plan is equal to the sum of the costs of all its actions. Therefore, the cost of an *I-plan* $\pi = \langle a_1, \ldots, a_n \rangle$ is given by $\Sigma_{i=1}^{n} c_{a_i}$.

In the case where all actions have the same cost, the cheapest path is also the shortest path. A task where each action has the same cost is a unit cost task. In this thesis, we focus on exactly these tasks.

2.2 PDDL

As mentioned above, a classical planning task consist of a domain and the task itself. We can formulate these two elements using the Planning Domain Definition Language (PDDL). This language is described by McDermott et al. [9] as follows. "PDDL is intended to express the "physics" of a domain, that is, what predicates there are, what actions are possible, what the structure of compound actions is, and what the effects of actions are." PDDL is a form of predicate logic, where we define the universe of the task, from which we can then ground the task and bring it into propositional logic form. We will now show an example of a domain as well as of a task and will use it throughout this thesis.

```
(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
               (ontable ?x)
               (clear ?x)
               (handempty)
               (holding ?x))
```

```
(:action pick-up
         :parameters (?x)
         :precondition (and (clear ?x) (ontable ?x) (handempty))
         :effect
         (and (not(ontable ?x))
              (not(clear ?x))
              (not(handempty))
              (holding ?x)))
(:action put-down
         :parameters (?x)
         :precondition (holding ?x)
         :effect
         (and (not(holding ?x))
              (clear ?x)
              (handempty)
              (ontable ?x)))
(:action stack
         :parameters (?x ?y)
         :precondition (and (holding ?x) (clear ?y))
         :effect
         (and (not(holding ?x))
              (not(clear ?y))
              (clear ?x)
              (handempty)
              (on ?x ?y)))
(:action unstack
         :parameters (?x ?y)
         :precondition (and (on ?x ?y) (clear ?x) (handempty))
         :effect
         (and (holding ?x)
              (clear ?y)
              (not(clear ?x))
              (not(handempty))
              (not(on ?x ?y)))))
```

Listing 2.1: PDDL blocks domain

The domain defined in Listing 2.1 describes a universe where blocks can be stacked. The predicates are variables which when grounded can either be true or false in each state, e.g. `handempty` would be true if the agent's hand is empty and false if he is holding a block. In addition, the domain defines the possible actions within the space. Each action has a set of preconditions which must hold for certain parameters so that the action can be executed. If an action is performed, its effects take place and result in the transition to a new state, where other variables are true or false.

Once the domain is defined, we can create tasks within the space.

```
(define (problem BLOCKS-4-1)
(:domain BLOCKS)
(:objects A B C D)
(:INIT (CLEAR B) (ONTABLE D) (ON B C)
       (ON C A) (ON A D) (HANDEMPTY))
(:goal (AND (ON D C) (ON C A) (ON A B))))
```

Listing 2.2: PDDL blocks task

A task is defined over explicit instances within the space. In this task, four objects are involved. The initial state is defined by `INIT`, which states which predicates are true in the beginning. The goal is again defined over specific predicates which have to be true in order to solve the task.

2.3 Search Algorithms

Planning tasks can be solved with search algorithms which explore the states in the tasks and determine the result, the plan, for a task. We will introduce two such search algorithms.

2.3.1 Blind Search

Blind search is a very naive approach to finding a solution. All states are explored with increasing distances from the initial state until a goal state has been found. A layer in blind search is determined by the g-value. This value indicates the distance of a state from the initial state I. This means that in unit cost tasks, all states which have a g-value of 1 are reachable with one action from the initial state. Blind search first expands all states which have a g-value of 1. Once all these states have been expanded, all states with a g-value of 2 will be expanded and so forth. The first layer in which a goal state is reached therefore marks the optimal cost of the task.

This algorithm is implemented with two lists, an *open list* and a *closed list*. The open list contains all states which can be expanded next, while the closed list contains the states which have already been expanded. Before a state s is expanded, we first check if it is already contained in the closed list. If s is already contained in the closed list and the g-value of the state in the closed list is lower or equal to the g-value of s, the state is not expanded again. However, if s was not yet expanded or has a lower g-value than the state in the closed list, s is expanded. During the expansion, all states which can be reached from s with cost 1 are added to the open list, while the state s is added to the closed list. At first, the open list contains only the initial state I, which is then removed and expanded. In the next step, the state which was inserted into the open list first is expanded. This process is continued until a goal state is removed from the open list.

Since this search expands all states per layer, it will find the cheapest path when considering unit cost tasks. However, due to this layered search, it also expands nodes which are

not goal-leading. Therefore, blind search is very slow and expensive. In order for a search algorithm to perform better, it should expand more promising nodes first.

2.3.2 A^* Search

The A^* search algorithm [5] belongs to the family of best-first algorithms. In each iteration, A^* will expand the most promising node. In order to determine which node is promising, we make use of the g-value and a heuristic value, the h-value. A heuristic estimates how far away a node is from the goal. There are many different heuristics, including the h^{max} heuristic, which will be used at a later point, and h^*, the perfect heuristic, which is defined as the actual distance of a state to its closest goal state. The A^* algorithm starts the search at the initial state I. All its successors are added to the *open list*. This list always contains the expansion candidates for the current step. In the first step, we evaluate which state s has the lowest value $f(s)$, where f is the evaluation function, which is defined as follows.

$$f(s) = g(s) + h(s),$$

where $g(s)$ is the distance from the initial state and $h(s)$ the heuristic value of the state s. When the most promising state s, i.e. the state with the lowest f-value in the open list, has been found, it is removed from the open list. If s is already contained in the closed list, we compare for the g-value as for blind search. In case the g-value of s, which was just removed from the open list, is higher, we do not expand s. Otherwise, we expand the state again and replace the state in the closed list. Before the expansion, it is first checked whether s is the goal state. If this is the case, a solution has been found and the path for the task is extracted. However, if the goal has not been found, all successors of s are added to the open list and s is added to the closed list. The algorithm continues in each step as described, by determining the most promising node and expanding it according to the f-value of the states.

As we expand only promising states, we expect to need fewer state expansions in order to find a goal than with blind search. This is especially true for tasks where the goal lies at a deeper layer or where layers are very broad. However, the performance of A^* always depends on the heuristic chosen.

When using an admissible heuristic, we can guarantee that our extracted plan is optimal [5]. A heuristic is called admissible if $h(s) \leq h^*(s)$ holds. This means that the heuristic value of a state is always lower than the perfect heuristic h^*, which maps a state s to the optimal cost of a plan for s.

2.4 Certificates

A certificate in general is an independent proof of a computed result of a planning task. For *I-plans*, it is rather simple to prove that it is indeed a solution, since this can be done

by executing the given plan. An existing tool to automatically validate a calculated plan is VAL [8].

A more complex task is to prove that a task is unsolvable. An unsolvability certificate aims to prove that no *I-plan* can exist for a given task.

A first solution was established by Eriksson et al. [2] using inductive certificates. An inductive set of states S has the property that any action a performed in a state $s \in S$ will lead to a state $s[a]$ which is again an element of S, so that once we enter S, we can never leave it. An inductive certificate for any state s is given by an inductive set S which contains s but none of the goal states. Therefore, an inductive certificate for a task Π is an inductive set S which contains I but no goal state. Since S is inductive, the goal states are unreachable from any state in S, in particular from I, which proves that Π is unsolvable.

A second approach was also developed by Eriksson et al. [3]. This time, the unsolvability certificates were created with the help of a knowledge base. The main idea is that the proof first establishes a knowledge base containing basic statements which are verifiable. This is followed by the application of derivation steps according to certain derivation rules. The goal of these derivations is to conclude from the generated statements that the task is unsolvable. In this case, a task is considered unsolvable if either I or all goal states are *dead*. A state is dead if it cannot be traversed by any plan. The derivation rules mainly help to determine further dead states and should show that either I or all goal steps are dead, proving that the task is unsolvable.

3 Reduction to Unsolvability

We want to reformulate a task such that it is unsolvable. We will do so by adding an upper bound for the cost which is lower than the optimal cost for the task. In order to reformulate an existing task such that an explicit maximum of costs is introduced, we have to modify the domain as well as the description of the task itself. In the following, we will explain how an existing domain and task have to be modified such that maximal costs are included. For this purpose, we will use the domain and task that were presented in Section 2.2. All changes are marked in red and further explained in the text.

Note that this transformation is only correct for unit tasks, where each action has a cost of 1. Implementation of different costs for each action will be discussed in Section 3.3.

3.1 Domain

The domain file has to be modified by adding predicates and modifying each action according to a certain schema. We have to define the two new predicates, `(cost ?c)` and `(next ?c ?n)`. The `cost` predicate defines the cost already used. With each action, `cost` should increase by one. Therefore, we have to introduce our `next` predicate. It defines the step size between each cost and thereby acts as a counter. This predicate is static, as no action changes its truth value. Therefore, any `next` predicate which is defined in the initial state will be true throughout all reachable states, while `next` predicates which have not been defined in the initial state will never be true for any reachable state.

We will now show how an action can be defined using the previously added predicates. The following action serves as an example. All modifications, marked in red, must be applied to all other actions as well.

```
(:action put-down
    :parameters (?x ?c ?n)
    :precondition (and (holding ?x) (cost ?c) (next ?c ?n))
    :effect (and (not(holding ?x))
                 (clear ?x)
                 (handempty)
                 (ontable ?x)
                 (cost ?n)
                 (not(cost ?c))))
```

Listing 3.1: PDDL action for unit cost

The required parameters are the current cost `c` as well as the next cost `n`. The two preconditions ensure that the given parameters have the properties of `c` being the current cost and `n` being the successor cost of `c`. The effect of an action a then has to include the updating of the cost. Here, it is important to note that updating the costs does not only include the new cost being `n` but also requires the old cost `c` to be false. This is necessary since in PDDL, like in first-order logic, the `(cost ?x)` predicate holds a truth value for every instantiation of `x`. Therefore, we have to set the old value to false since both `(cost c)` and `(cost n)` would otherwise be true, and this would impair our counting.

3.2 Task

In order to introduce the previously defined costs into our specific task, we have to expand the objects and modify the start configuration `INIT`. The additional objects as well as the statements for the initial state depend on the maximum cost allowed, which is denoted by x. The objects that have to be introduced are all consecutive numbers from 0 to x.

The required statements for the initial state include the cost used at the beginning, namely zero. In addition, we need to define the static steps, using the `next` predicate for each step from 0 up to x.

```
(define (problem BLOCKS-4-1)
(:domain BLOCKS)
(:objects A B C D 0 1 2 3 4 5)
(:INIT (CLEAR B) (ONTABLE D) (ON B C) (ON C A) (ON A D)
       (HANDEMPTY) (COST 0) (NEXT 0 1) (NEXT 1 2)
       (NEXT 2 3) (NEXT 3 4) (NEXT 4 5))
(:goal (AND (ON D C) (ON C A) (ON A B)))
)
```

Listing 3.2: PDDL task with cost

This extract is an example of a task where the initial cost is zero, the size of the unit cost is one and the maximal cost x is 5. The cheapest solution for the initial task which has been computed is 6. Therefore, the task defined above, where the solution is limited to cost 5, would obviously be unsolvable.

Defining a task as described allows for solutions with any costs between 0 and x. If instead only a specific cost y with $y \in \{0, .., x\}$ should be considered, we can include this in the goal statements with the statement `(cost y)`.

3.3 Individual Cost

In this section, we will present possible modifications for tasks where not all actions have the same cost. There are two general approaches to this.

The first approach does not need any additional predicates. Instead, it uses the `next` predicate already introduced. If the step size of `next` is one, we can modify the cost of a single action by introducing additional parameters and preconditions in the following form.

```
(:action put-down
    :parameters (?x ?c ?c1 ?c2 ... ?cn-1 ?cn)
    :precondition (and (holding ?x) (cost ?c) (next ?c ?c1)
                       (next ?c1 ?c2) ... (next ?cn-1 ?cn))
    :effect (and (not(holding ?x))
                 (clear ?x)
                 (handempty)
                 (ontable ?x)
                 (cost ?cn)
                 (not(cost ?c))))
```

Listing 3.3: PDDL action for cost n

This solution introduces a sequence of numbers $c, c_1, c_2, ..., c_{n-1}, c_n$ where n is the cost of the action, assuming that `next` is always defined for two adjacent numbers. In fact, we have to check that `next` is true for all the numbers in the sequence to ensure that `cn` is larger than `c` by value `n`. Again, we have to update our cost by removing the old cost and defining the new cost `cn`. The idea behind this solution is rather simple, but infeasible for tasks where the difference between costs of actions is large. This is due to the fact that `next` would have to be defined for a small step size so that actions with small costs can be realized. On the other hand, we would need a long sequence of adjacent numbers, hence numerous parameters and preconditions to model the costs of an expensive action.

The second approach is to replace the `next` predicate by multiple predicates for different step sizes. Here, once the predicates are defined, it is quite easy to modify the actions. We can just modify Listing 3.1 by replacing the precondition `(next ?c ?n)` with the suitable step size predicate. This version is very simple in the domain, but we need to define all possible steps as static statements in the initial state of our task. This can again be infeasible for tasks with highly varying costs.

It is also possible to combine the two approaches. For example, one could introduce a predicate for step size one `(one ?c ?n)` as well as for step size ten `(ten ?c ?n)`. The parameters and preconditions of a task could then be used to first increase the cost in steps of ten and then in steps of one.

```
(:action put-down
    :parameters (?x ?c ?t1 ?t2 ?o1 ?o2 ?n)
    :precondition (and (holding ?x) (cost ?c)
                       (ten ?c ?t1) (ten ?t1 ?t2)
                       (one ?t2 ?o1) (one ?o2 ?o2) (one ?o2 ?n))
    :effect (and (not(holding ?x))
                 (clear ?x)
                 (handempty)
                 (ontable ?x)
                 (cost ?n)
                 (not(cost ?c))))
```

Listing 3.4: PDDL action for cost 23

However, this only simplifies notation in the domain, but we still have to define `one` and `ten` for every possible pair of numbers in the initial state of the task.

Introducing actions which do not have any cost is fairly simple. In this case, the action does not have to be modified at all, i.e. no counting parameters, preconditions or effects are required. This is due to the fact that an action with cost zero has no influence at all on the `cost` predicate.

4 Experiments

In determining the costs for a large number of solvable unit cost tasks, we can make use of the reformulations for unit cost tasks as described in Chapter 3. We can then run a search on an unsolvable task, create certificates for unsolvability and verify them.

The calculations were performed at the sciCORE[1] scientific computing center at the University of Basel. The experiments were run on a cluster consisting of an Intel Xeon E5-2660 CPU with 2.2 GHz. For the search, including the creation of the certificates, a time limit of 30 minutes as well as a memory limit of 3584 MiB were used. The verification of the certificate had a limit of 4 hours and a memory restriction of 2000 MiB.

The general framework for the experiments was Downward Lab [10]. It was used to run the Fast Downward 21.12 [6] implementation. We used A^* search with the h^{LM-cut} heuristic [7] to initially solve the tasks. We then extracted the cost for each task and modified all unit cost tasks for which an optimal plan could be found. The resulting tasks were then run again with a fork of Fast Downward that implemented certificates for unsolvability[2], introduced by Eriksson et al. [3]. We used h^{max} as well as $h^{M\&S}$ as heuristics for these runs which showed that the modified tasks are unsolvable. The verifier and proof-generating planning algorithms [3] were used to verify that the tasks are unsolvable, which proved that the cost which was calculated in the first step was indeed optimal.

4.1 Results

For the results, we compare different attributes of our three runs. The three runs are the initial run on the original task using h^{LM-cut} as well as the two runs on the modified tasks using h^{max} and $h^{M\&S}$. First, we will discuss how successful the runs on the modified tasks were. We will do so by considering the number of created and verified certificates, as well as the number of failed runs. We will then compare the time difference between the run on the original tasks and the runs on the cost restricted tasks. We will also compare the number

[1] http://scicore.unibas.ch/
[2] https://github.com/salome-eriksson/downward-unsolvability

of expanded states as well as the time needed for the two runs on the modified tasks using h^{max} and $h^{M\&S}$. Finally, we will compare the memory used by the different runs during the search. The results presented are only an excerpt from the data and figures generated during the experiments. More figures can be found in Appendix A.

universe (number of tasks)	created		verified	
	h^{max}	$h^{M\&S}$	h^{max}	$h^{M\&S}$
airport (28)	16	11	14	11
blocks (28)	15	18	15	18
depot (7)	2	4	2	2
driverlog (13)	5	9	4	7
freecell (15)	7	14	7	13
grid (2)	1	1	1	1
gripper (7)	5	6	5	5
hiking-opt14-strips (9)	6	9	6	8
logistics00 (20)	10	10	10	10
logistics98 (6)	1	2	1	2
miconic (141)	40	45	40	40
movie (30)	30	30	30	30
mprime (22)	15	22	13	20
mystery (17)	12	15	10	14
openstacks-strips (7)	7	7	7	7
organic-synthesis-opt18-strips (7)	7	7	7	7
pipesworld-notankage (17)	10	12	8	12
pipesworld-tankage (12)	6	12	6	10
psr-small (49)	43	49	42	47
rovers (8)	4	4	4	4
satellite (7)	4	4	4	4
snake-opt18-strips (7)	2	0	2	0
storage (15)	13	13	12	13
termes-opt18-strips (6)	1	4	1	3
tidybot-opt11-strips (14)	3	1	3	1
tidybot-opt14-strips (9)	0	0	0	0
tpp (7)	5	6	5	5
trucks-strips (10)	4	4	3	3
visitall-opt11-strips (11)	8	9	8	8
visitall-opt14-strips (5)	2	3	1	2
zenotravel (13)	8	8	7	8
total (549)	**292**	**338**	**278**	**315**

Table 4.1: Number of created and verified certificates per domain for h^{max} and $h^{M\&S}$ in comparison to total number of runs

As Table 4.1 shows, $h^{M\&S}$ was generally more successful when creating and verifying certificates. For about 62% of the tasks, a certificate could be created and about 93% of these certificates could be verified. In contrast, h^{max} generated a certificate for only about 53% of the tasks. However, 95% of the generated certificates could be verified, slightly more than for $h^{M\&S}$.

Using the data from Table 4.2, we can retrace the errors which occurred during the runs. The sum of the errors added to the number of tasks for which a certificate could be created adds up to 549, which is the total number of tasks.

error	h^{max}	$h^{M\&S}$
search out of memory	5	155
search out of time	230	34
segfault	5	5
translate out of memory	17	17
total	**257**	**211**

Table 4.2: Number of total errors for h^{max} and $h^{M\&S}$

We can observe that for h^{max}, most of the errors were due to the time restriction for the task. Since the search could not be finished, no certificate was generated. This can be explained by the fact that $h^{M\&S}$ provides a better heuristic estimate than h^{max}. For unsolvable tasks, a good heuristic guidance is crucial because it might detect at an early stage that the task is unsolvable. In contrast, a heuristic which cannot detect early on that the task is unsolvable requires the expansion of more states and hence more time.

In contrast to runs using h^{max}, most of the failed runs for $h^{M\&S}$ are due to the lack of memory. The reason may be that $h^{M\&S}$ needs preprocessing, which is performed prior to the search. For large tasks, the additional memory needed for these calculations combined with the actual search might exceed the memory limit.

The few segfaults which occurred during the experiment are cases where there was not enough memory for the translation. These cases are not listed as *translate out of memory* as there was not enough memory left to gracefully end the experiment. Since the run could not be ended properly, a segfault occurred instead of a translate out of memory error.

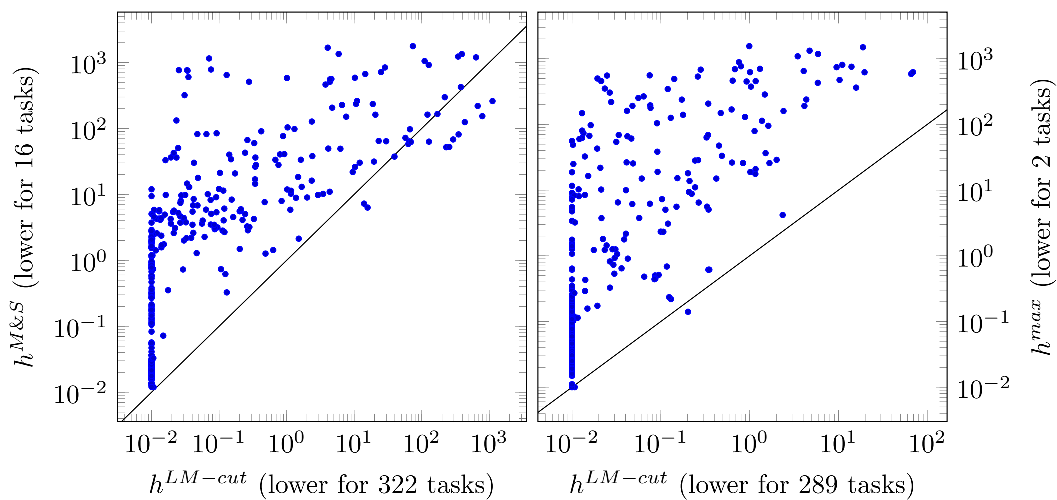

Figure 4.1: Time comparison between h^{LM-cut} and $h^{M\&S}$ (left) and between h^{LM-cut} and h^{max} (right)

Figure 4.1 shows that, in general, using h^{LM-cut} on the original task is faster than using h^{max} and $h^{M\&S}$ on the modified task. For modified tasks, certificates are created in addition to the search, which takes additional time. The search itself may also take longer, as all possible paths have to be explored in order to conclude that the problem is unsolvable. In contrast, we can stop the search with h^{LM-cut} as soon as one solution has been found.

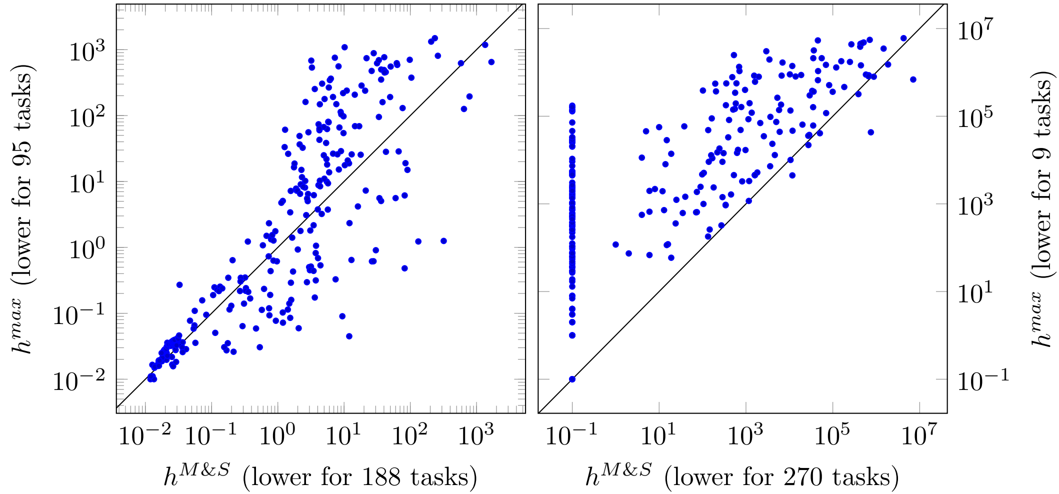

Figure 4.2: Time comparison (left) and expansion comparison (right) between $h^{M\&S}$ and h^{max}

In Figure 4.2 (left), we can observe that tasks which took less time to solve had a tendency to be solved faster when using h^{max}. In contrast, tasks, which took rather long to solve in general, were mostly solved faster by $h^{M\&S}$. Since $h^{M\&S}$ needs preprocessing, additional time has to be taken into consideration. For easy tasks, these expensive calculations prior to the search are not compensated by a faster search. However, for more difficult tasks $h^{M\&S}$ performs better than h^{max} since it provides better heuristic guidance.

In Figure 4.2 (right) we can clearly see that h^{max} needs more expansions during the search than $h^{M\&S}$. In some cases, $h^{M\&S}$ does not even need to make expansions because it detects already in the initial state that the task is unsolvable. These are exactly the states with an expansion value of 10^{-1} for $h^{M\&S}$.

In Figure 4.3, we can see that the run using h^{LM-cut} has a significantly lower memory consumption than the runs with h^{max} and $h^{M\&S}$. As mentioned above, this is due to the fact that when trying to solve an unsolvable task, all possible paths have to be explored and hence more memory is needed. In the first run, however, the search can be stopped once a solution has been found. The certificate which is created alongside the runs with h^{max} and $h^{M\&S}$ also increases the amount of memory used.

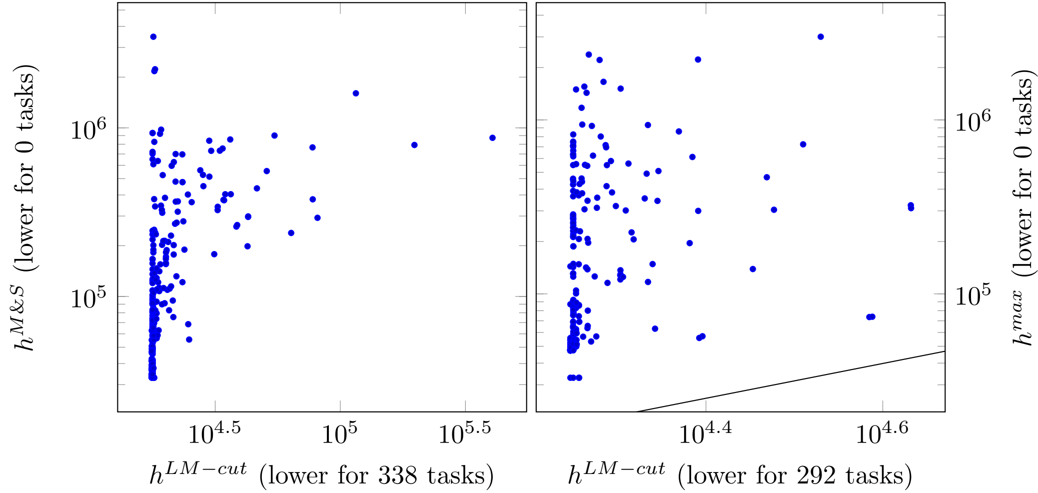

Figure 4.3: Memory comparison between h^{LM-cut} and $h^{M\&S}$ and between h^{LM-cut} and h^{max}

By comparing the left and right sides of Figure 4.3, we can observe that $h^{M\&S}$ tends to need less memory than h^{max} (see also Figure A.1). This is coherent with the observations from Figure 4.1 and Figure 4.2. Since $h^{M\&S}$ tends to need fewer expansions and also has a slight tendency of needing less time during the run, it seems natural that the memory needed is lower as well.

In order to verify that unsolvable tasks need more expansions, we had to run the modified tasks again using the same configuration as for the initial run. We could not use the results from the runs with h^{max} and $h^{M\&S}$ for this comparison, as each heuristic performs differently on the same task. Therefore, we used the same configuration, namely h^{LM-cut}, for both runs. We compared the number of expanded states from the run on the modified tasks to the number of expanded states in the initial run on the original tasks.

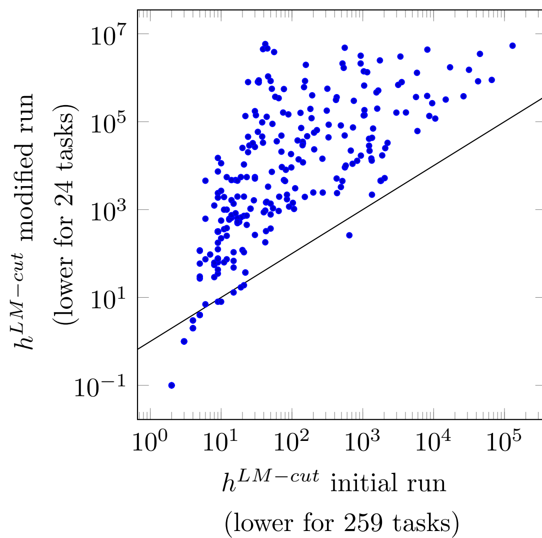

Figure 4.4: Expansions comparison between initial and modified run for h^{LM-cut}

With the results from Figure 4.4, we can verify that our statement about the difference in the number of expanded states is true. During the first run on the initial task, fewer expansions were needed in order to find a goal and complete the search. However, for the modified run which tried to solve the unsolvable modified task, more expansions were needed in order to conclude that the task in unsolvable and to finish the search. Comparisons between the number of expanded states for the initial run and the two runs using $h^{M\&S}$ and h^{max} can be found in Appendix A.

The few outliers for which the modified run needed fewer expansions could be due to the fact that h^{LM-cut} may detect that all states which have a distance of $x-1$, where x is the optimal cost, from the initial state are dead ends. In this case, no further states will be explored, whereas in the initial search more states have to be expanded in order to reach the goal.

4.2 Conclusion

We were able to verify unsolvability for the modified tasks in 62% of the cases for $h^{M\&S}$ and in 53% of the cases for h^{max}. Most of the certificates that were generated could also be verified. The modified tasks could be solved using a moderate amount of memory and time. In general, $h^{M\&S}$ performed better on the modified tasks than h^{max}, because it provides better heuristic guidance.

Even though the results of our experiments seem good, it remains unclear if some tasks are unsolvable due to possible errors in the transformation of the tasks. The transformation is prone to error, as different tasks may require different transformations. Since the translation of the task itself is not verified, errors in the task itself cannot be detected. Therefore, a solution independent of the transformation of tasks would be desirable. Such a solution will be offered in Chapter 5 by creating a proof system for the optimal cost of a planning task based on the proof system for unsolvable planning tasks introduced by Eriksson et al. [3].

5

Proof System for Optimality

The goal of optimality certificates is to prove that a computed plan for a task is optimal. The certificate is constructed alongside the search and verified independently. The aim of this chapter is to introduce such certificates for optimal solutions. As before, we will consider only tasks with unit cost actions, where all actions of the task have exactly cost one.

In the following, we will introduce an approach using sets of states which allows us to make statements about the optimal cost of a task. We will build sets which require a certain minimal cost to a goal. These sets will be built iteratively until we are able to make a statement about the minimal cost from the initial state to a goal. This minimal cost should correspond to the optimal cost of the task computed.

5.1 General Concept

We can validate that the optimal solution of a task has cost x by showing that the task has no solution with cost $x-1$ or lower, and that there is a solution with cost x. In order to show this, we can build sets of states from which a goal can be reached with at least cost c. We will create sets for all possible minimal costs $c \in \{0, ..., x\}$ for reaching the goal. We define a set S_x as a set of states which need at least cost x to reach a goal state.

The set S_0 is equal to S_{All}, the set of all states of the state space, since all states have at least distance 0 to a goal. We know that S_1 must contain only states which need one action and hence cost 1 to a state in S_0. We can create such sets for all minimum distances to the goal up to the optimal cost x. All sets S_x always have the form $S_x = \{s \mid s[a] \in S_{x-1} \text{ for all actions } a\}$.

In order to show that if a task is solvable, it requires at least cost x, we can use the fact that if S_x contains the initial state, then the minimal optimal cost is at least x. If we want to prove that x is indeed the optimal cost, we can use the generated plan with cost x as a witness that a plan with cost x exists.

We have now explained the general concept behind the certificate. In a next step, we will formally define how such a certificate can be built and how the specified sets of states can be generated.

5.2 Formal Proof System

This proof system makes specific statements over sets of states, which then lead to the conclusion of the proof. These sets of states can be constants such as the set containing the initial state $\{I\}$, the set of goal states S_G or the set of all states S_{All}. The state sets can also be set variables, which are sets that are described in a specific formalism such as BDDs or concrete state enumeration as well as the complement, union, intersection, predecessors or successors of state sets.

During the proof, the state sets are defined first. Later, basic statements and derivation rules are applied. The basic statements always need to be verified individually, while the derivation consists of a sequence of templates, so-called derivation rules, which are applied to concrete state sets during the proof. The fact that these derivation rules are true for any concrete state set is shown outside the proof. In a proof step, either a basic statement or a derivation rule is applied. The consecutive application of these steps leads to the derivation which proves the optimal cost of a task.

We will start by introducing the derivation rules necessary for this proof system.

5.2.1 Derivation Rules

This proof system is based on x-states, states which require at least cost x to reach the goal.

Definition 1 (x-state)**.** *A state is an x-state if all paths from the state to a goal have at least cost x. Therefore, all states are 0-states. An x-state is also a y-state if $x > y \geq 0$.*
A set of states is called x-state set if it consists only of x-states. Therefore, the set of all states S_{All} is a 0-state set. If a state set is an x-state set, and $x > y \geq 0$, then it is also a y-state set.

We can use this property to make a statement about the union of different state sets. The union of arbitrary x-state sets with different x is itself an y-state set, where y is the smallest value for x among the x-state sets.

We will now define the rules which follow from this definition.

- *S_{All} is a 0-state set.*
- *Any set of states is a subset of S_{All}.*
- *A subset of an x-state set is an x-state set.*
- *The union of arbitrary x-state sets with same x is an x-state set.*

- *The union of arbitrary x-state sets with differing x is a min(x)-state set.*

We need to introduce a rule which allows us to use the external knowledge that the cost of the generated plan is x. This rule needs to be verified individually outside the proof system. In order to prove this statement, a verifier such as VAL can be used. We will formulate the statement for our proof system as follows.

- *The cost of the generated plan is x.*

The following rules define how the x-states are built and are therefore the core of the system.

Theorem 1. *Let S be a set of states and S_x an x-state set, such that all successors of S are contained in S_x, i.e., $S[A] \subseteq S_x$. If $S \cap S_G = \emptyset$, then S has at least distance $x + 1$ to the goal and is therefore an $(x + 1)$-state set.*

Proof. According to the definition, a state s is an x-state if it has at least distance x to a goal state. Since all successors of S lie in S_x, we need cost 1 to reach S_x from S. As we have at least distance x to the goal from S_x and S does not contain a goal state, we can conclude that we must have at least distance $x + 1$ from the states in S to a goal state, since the path trough S_x is the only option to reach a goal. This distance cannot be lower because all successors of S are in S_x, i.e. we cannot leave S via another state set with lower distance to the goal. □

The following rules define how the minimal cost of a task can be determined.

Theorem 2. *Let S_x be an x-state set. If $\{I\} \subseteq S_x$ and the generated plan has cost x, then the optimal cost for the task must be exactly x.*

Proof. Since S_x contains only states which have at least distance x to the goal and the initial state is contained in S_x, we know that a solution of the task must have at least cost x. If the cost of the generated plan is also x, we can use the plan as a witness that there is a solution which has exactly cost x. We can therefore conclude that the optimal cost of the plan is indeed x. □

We will now summarize all rules which have been introduced so far in the following definition.

Definition 2 (derivation rules)**.**

D1 S_{All} *is a 0-state set.*

D2 $S \subseteq S_{All}$

D3 If S_x is an x-state set and $S \subseteq S_x$, then S is an x-state set.

D4 If $S_1, \ldots, S_n$ are x_i-state sets and $x_i = x_j$ for every $i, j \in \{1, \ldots n\}$, then $\bigcup_{i \in \{1,\ldots,n\}} S_i$ is an x-state set

D5 If $S_1, \ldots, S_n$ are x_i-state sets and $x_i \neq x_j$ for every $i, j \in \{1, \ldots n\}$, then $\bigcup_{i \in \{1,\ldots,n\}} S_i$ is a min(x)-state set

D6 The cost of the generated plan is x.

D7 If S_x is an x-state set, $S[A] \subseteq S_x$ and $S \cap S_G \subseteq \emptyset$, then S is an (x+1)-state set.

D8 If S_x is an x-state set, $\{I\} \subseteq S_x$ and x is the cost of the generated plan, then the optimal solution has cost x.

These derivation rules define the basis for the actual proof. In the derivation steps, the placeholder sets like S_x or S in the rules are replaced by sets of states. As mentioned above, such a set can be either a constant set, a set variable or their concatenation, intersection, complement, predecessors as well as successors.

The derivation steps are applied to a knowledge base. This knowledge base consists only of atomic statements such as "S is an x-state set", "$S \subseteq S'$" as well as "The cost of the generated plan is x" and "The optimal solution has cost x". Again, S and S' are replaced by set expressions. If all premises of a derivation step have been derived already, we can apply the step. This results in adding its consequences to the knowledge base. The special statement "the optimal solution has cost x" concludes the proof.

During the proof, the set expressions in the derivation rules are not interpreted, i.e. the rules are applied on the syntactic level without regard to the interpretation of their elements.

5.2.2 Basic Statements

In order to be able to apply our defined derivation rules, we always need an initial knowledge base from which we can then make further derivations. These derivations will then be task specific, even though the derivation rules themselves are not, since the placeholder sets in the specified rules are replaced by sets which fulfil the conditions for the provided task.

In the proof, we will create our initial knowledge base by making use of basic statements as well as derivation rules which have no premises. This step is followed by the application of further basic statements and derivation rules. In the derivation, the derivation rules are simply applied on a syntactic level where we do not consider concrete sets but only abstract expressions of them. In addition, a rule can be applied only if all its premises have been established either by basic statements or previous derivations. Therefore, the order of the derivation steps matters.

Since our derivation rules require the same sort of statements as the unsolvability proof system [3], we can take their definition of basic statements. As stated, the statements of the form "S is an x-state set" and "$S \subseteq S'$" build our knowledge base. Again, S and S' are set expressions. However, we will allow only a subset of these statements as basic statements. This is because we want our basic statements to be efficiently verifiable so that our proof itself is efficiently verifiable as well. We will include only basic statements which cannot be derived from other basic statements. We will allow only statements of the form "$S \subseteq S'$", where S and S' are allowed to be replaced only by specific set expressions. In the following, we define the permitted basic statements. We will use X, X' and X'' as set variables. Further, we will use L and L' as atomic set term literals, which include set

variables X, constant sets such as the set of goals, the set of the initial state or the empty set as well as the complements of these.

Definition 3 (basic statements)**.**

B1 $L \subseteq L'$

B2 $X \subseteq X' \cup X''$

B3 $L \cap S_G \subseteq L'$

B4 $X[A] \subseteq X \cup L$

The basic statements in combination with the derivation rules allow us to define the structure of a proof. We will sketch such a proof in the following chapter.

6

Certificates for Optimality

In this chapter, we will sketch the general form of a proof for different search algorithms. This proof can be used as a certificate for the optimal cost of a task. We will use the basic statements from Definition 3 as well as the derivation rules from Definition 2. The general syntax of steps in such a sketch is as follows.

(m) $DX, \{(a), \dots\} \rightarrow s$

(n) $BY \rightarrow s'$

We distinguish between steps where a derivation rule (DX) is applied and steps where a basic statement (BY) is applied. In both cases, we first state the number of the current step, here denoted by (m) and (n), respectively.

If the mth step consists of applying a derivation rule, we note the applied rule by DX, where X is the number of the corresponding rule. As a second argument, we state a set of previously applied steps $(a), \dots$ whose derived statements are needed as premises for the current step. If no premises are needed for the derivation rule, we can omit this argument. The statement s is the statement which we can conclude in the current step.

If the nth step consists of applying a basic statement, we note the applied rule by BY, where Y is the number of the corresponding rule. As we do not need any premises for basic statements, we can simply state the statement s' which we want to verify in the current step.

We will apply the previously defined rules for different kinds of search algorithms. Depending on the algorithm, more intermediate steps are necessary. However, the overall structure is similar for every algorithm. We can create certificates using the predefined rules for any unit cost task. However, depending on the search algorithm used, the specified certificate may not be efficient.

In order to apply the proof steps, we need to define the concrete sets for the corresponding proof. The creation of these sets will again differ slightly from algorithm to algorithm.

We will now discuss how the concrete sets of states $S_1, \dots, S_x$ as well as the general proof are built for different search algorithms.

6.1 Blind Search

To create a certificate for blind search (see Section 2.3.1), we need to build concrete sets which have at least a certain distance to a goal. As before, we denote a set of states which have at least distance 1 to the goal with S_1. The 0-state set S_{All} simply consists of all states of the task. In order to build the sets $S_1, \dots, S_x$, we will need to make use of the g-value. We know that a goal state must have a g-value of at least x, as x is the optimal cost for the task. From this, we can derive that all states which have a g-value of $x-1$ or less must have at least distance 1 to the goal. We can therefore define S_1 as the set of all explored states which have a g-value between 0 and $x-1$. The set S_2 would then be the set of explored states which have a g-value between 0 and $x-2$ and so forth.

Since x is the optimal cost for the task, we know that there must be at least one state in S_1 for which the shortest path to the goal has exactly cost 1, namely the state prior to the goal state in the path with optimal cost. In fact, there must always exist a state in S_n for which the optimal path to the goal has exactly cost n for all sets S_n with $n \in \{0, \dots, x\}$. We can guarantee this by making use of the same reasoning as before. We can therefore also conclude that I must be in the set S_x, since the initial state must have the optimal distance of x to the goal state.

6.1.1 Proof of Premises for Blind Search

We will now prove some observations about blind search, which are necessary for the creation of the corresponding certificate.

Theorem 3. *$S_n \cap S_G = \emptyset$ for any set S_n with $n \geq 1$.*

Proof. Blind search expands in order of g-value and stops when finding a goal. Since blind search stops when finding a goal with g-value x, we know that all states with g-value $y < x$ cannot be goals. Therefore, the g-value of a goal state must be at least x. Since the sets S_n consist only of states with g-value $x-n$ and $n \geq 1$, we can conclude that no goal state can be contained in any such set. □

Theorem 4. *$S_n[A] \subseteq S_{n-1}$ for any sets S_n, S_{n-1} with $n \geq 1$.*

Proof. Since we consider only unit cost tasks, we know that the successor s' of a state s can have a g-value of at most $g(s)+1$ (it can also be less than $g(s)+1$ since there might be a shorter path from I to s' that does not lead over s). Since S_n consists of states which have a g-value of at most $x-n$, we know that all successors of S_n can have a g-value of at most $x-n+1 = x-(n-1)$. Since we have seen all states in S_n, we know that all successors have to be either in the open or in the closed list and are therefore known. It follows that all successor states of S_n are known and have a g-value of at most $x-(n-1)$. These states are exactly the states contained in S_{n-1}. □

6.1.2 Proof Sketch for Blind Search

The general idea of the proof is to start with the set of all states and the fact that this set is a 0-state set. In a next step, we consider the set S_1. As proven, this set does not contain any goal state, and all its successors are in the set S_{All}. We can therefore conclude that the set S_1 is a 1-state set. We continue in the same manner for the remaining sets $S_2, \ldots S_x$ by always applying the same derivation rules. Once we have concluded that S_x is a x state set, we know that all states in S_x have at least distance x to the goal. We also know that x is the cost of the generated plan. If the initial state I is contained in the set S_x, we can conclude that the optimal solution has indeed cost x.

We will now provide a formal sketch of the proof. The sets $S_1, \ldots, S_x$ will be used as defined for blind search, where x is the optimal cost of the task.

(1) $D1 \rightarrow S_{All}$ is a 0-state set

(2) $B4 \rightarrow S_1[A] \subseteq S_{All}$ [3]

(3) $B3 \rightarrow S_1 \cap S_G \subseteq \emptyset$

(4) $D7, \{(1), (2), (3)\} \rightarrow S_1$ is a 1-state set

(5) $B4 \rightarrow S_2[A] \subseteq S_1$

(6) $B3 \rightarrow S_2 \cap S_G \subseteq \emptyset$

(7) $D7, \{(4), (5), (6)\} \rightarrow S_2$ is a 2-state set

...

(n-3) $D7, \{$(n-6), (n-5), (n-4)$\} \rightarrow S_x$ is a x-state set

(n-2) $B1 \rightarrow \{I\} \subseteq S_x$

(n-1) $D6 \rightarrow$ The cost of the generated plan is x.

(n) $D8, \{$(n-3), (n-2), (n-1)$\} \rightarrow$ The optimal solution has cost x.

6.2 A^* Search

In contrast to the blind search, A^* search (see Section 2.3.2) uses a heuristic as an indicator of how close a state is to the goal. This h-value is used in addition to the g-value to determine which state is to be expanded next during the search. We can make use of different heuristics, which have distinct attributes. We will first describe the general form of the proof for A^* search, and in a later step include proofs for the h^{max} heuristic.

[3] Technically, $B4$ has the form $X[A] \subseteq X' \cup L$. We could conform to this by writing $S_1[A] \subseteq S_all \cup \emptyset$. But for the sake of simplicity, we will omit the union with the empty set.

To create a certificate for A^* search, we again need to build concrete sets. We once more use the notation of S_x sets introduced above. The 0-state set S_{All} again consists of all states of the task. We will distinguish between states which have been expanded during the search and states which have not. States which have been expanded have only successors that are known. For these states, we again use the g-value to determine to which x-state set they belong, as already explained for blind search. This means that an expanded state with a g-value of $x - n$ or lower will be assigned to the set S_n.

Since we require all successors of an n-state set to be in an $(n-1)$-state set, we cannot simply add non-expanded states to an n-state set, since their successors are not in an $(n-1)$-state set. Therefore, we will use the heuristic value h of the non-expanded states to prove that they are h-states so that we can form unions over any n-state set and h-state sets for which $h \geq n$ holds. Once this union has been formed, our premise that all successors of a state are in the corresponding state-set is fulfilled. We will prove individually for the h^{max} heuristic that a state with a heuristic value of h is actually an h-state. In general, this can be proven for other admissible heuristics as well.

We know that the optimal path of the cost must consist of states which have been expanded during the search. Since we use the same assignment for expanded states to sets as in blind search, we can apply the same argumentation as in Section 6.1 in order to verify that S_x contains the initial state I and that all other states of the optimal path are contained in their respective sets as well.

6.2.1 Proof of Premises for A^* Search

As for blind search, we will first prove some observations about A^* search, which we will need during the creation of the certificate.

Theorem 5. *$S_n \cap S_G = \emptyset$ for any set S_n with $n \geq 1$.*

Proof. Since S_n contains only states which have been expanded and have a g-value between 0 and $x - n$, we can use the same argumentation as in Section 6.1.1. □

Theorem 6. *$S_n[A] \subseteq S_{n-1} \cup \bigcup_{h(s) \geq n-1} \{s\}$ for any sets S_n, S_{n-1} with $n \geq 1$.*

Proof. For any state in S_n, its expanded successors are in S_{n-1} as explained in Section 6.1.1. The non-expanded states are in $\bigcup_{h(s) \geq n-1} \{s\}$. We will prove that all non-expanded successors of S_n must indeed have a h-value which is equal or larger than $n-1$. We will show this property using a contradiction.
Assume the state s' is a non-expanded successor of a state $s \in S_n$ and $h(s') < n-1$. The following three statements follow.

I $g(s) \leq x - n$ since $s \in S_n$

II $g(s') \leq x - n + 1$ since $s' = s[a]$ and $c_a = 1$

III $h(s') < n - 1$ by assumption

From these statements, we can conclude that $f(s') = g(s') + h(s') < x - n + 1 + n - 1 = x$. This result means that the heuristic value of s' is lower than the optimal cost of the task x. Since s' must be in the open list and A^* always expands states with a lower f-value first, the state s' must have been expanded before a goal state s_g since $f(s_g) = x$ for goal-aware heuristics. Since we consider admissible heuristics which are always goal-aware, this statement holds for our case. However, the statement is a contradiction to our assumption, as we assumed that s' is a non-expanded state.

We can therefore conclude that no non-expanded successor of a state in S_n can have a h-value which is smaller than $n-1$. Therefore, all non-successors s of S_n must be contained in $\bigcup_{h(s) \geq n-1} \{s\}$. □

6.2.2 Proof Sketch for A^* Search

The general idea of the proof is to first derive for each non-expanded state that it is an h-state so that we can later use this fact to form the union with other x-state sets. As for blind search, we start with the set of all states and derive that the set S_1 is a 1-state set. In contrast to blind search, we cannot simply continue with the set S_2. We first have to form the union over the set S_1 and all non-expanded states, which are h-states where h is equal or larger to 1. We can conclude that this union must be a 1-state set and also includes all successors of the states in S_2. We can now continue by deriving the fact that S_2 is a 2-state set. We continue the derivation by always forming the union over an x-state set, with all expanded states which have a h-value larger or equal to x. Once we have derived the fact that I is an element of an x-state set, x being the cost of the generated plan, we can conclude in the same manner as for blind search that x is the optimal solution of the task.

We will now provide a formal sketch of the proof. The sets $S1, \dots, S_x$ are used as defined for A^* search. In addition, we have the sets consisting of a non-expanded state $s_{i,j}$, where i is the h-value of the state and j distinguishes between states with the same h-value. Again, x is the optimal cost of the task.

(1_1) $h(s_{1,1}) = 1 \rightarrow \{s_{1,1}\}$ is a 1-state set

...

(1_n) $h(s_{1,n}) = 1 \rightarrow \{s_{1,n}\}$ is a 1-state set

...

(k_1) $h(s_{k,1}) = k \rightarrow \{s_{k,1}\}$ is a k-state set

...

(k_m) $h(s_{k,m}) = k \rightarrow \{s_{k,m}\}$ is a k-state set

(1) $D1 \rightarrow S_{All}$ is a 0-state set

(2) $B4 \rightarrow S_1[A] \subseteq S_{All}$

(3) $B3 \rightarrow S_1 \cap S_G \subseteq \emptyset$

(4) $D7, \{(1),(2),(3)\} \rightarrow S_1$ is a 1-state set

(5) $D5, \{(1_1),\dots(1_n),\dots,(k_1),\dots,(k_m),(4)\} \rightarrow S_1 \cup \bigcup_{h(s)\geq 1}\{s\}$ is a 1-state set.

(6) $B4 \rightarrow S_2[A] \subseteq S_1 \cup \bigcup_{h(s)\geq 1}\{s\}$ [4]

(7) $B3 \rightarrow S_2 \cap S_G \subseteq \emptyset$

(8) $D7, \{(5),(6),(7)\} \rightarrow S_2$ is a 2-state set

...

(n-3) $D7, \{$(n-6), (n-5), (n-4)$\} \rightarrow S_x$ is an x-state set

(n-2) $B1 \rightarrow \{I\} \subseteq S_X$

(n-1) $D6 \rightarrow$ The cost of the generated plan is x.

(n) $D8, \{$(n-3), (n-2), (n-1)$\} \rightarrow$ The optimal solution has cost x.

6.3 h^{max} Heuristic

We will first introduce the h^{max} heuristic so that we can later use this definition to prove that any non-expanded state with a heuristic value of h is an h-state. As h^{max} is an admissible heuristic, we are able to do so. The definition for the h^{max} heuristic of a state s is taken from Bonet and Geffner [1].

$$h^{max}(s) = min(\max_{v\in pre(G)}(g_s(v)) + c_a) = min(\max_{v\in pre(G)}(g_s(v)) + 1),$$

where c_a is the cost of the action a which results in G. Since we consider only unit cost tasks, c_a will always be equal to 1. The value $g_s(v)$ is the cost for starting in state s and achieving variable v, which is a precondition needed to reach the goal G. This cost is calculated iteratively by starting with the variables v_I in state I for which $g_s(v_I) = 0$. For variables v' which can be achieved by applying an action that is coherent with the available preconditions, $g_s(v_1) = min(max(g_s(v_0)) + c_a)$ holds. The variable v_0 is a precondition of the action a which results in variable v_1. Therefore, the g_s value of a variable v always corresponds to the cheapest sum of the cost of an action resulting in v plus its most expensive precondition.

This heuristic estimates the distance to the goal by determining the cost of the precondition which is most expensive to achieve. Since h^{max} is an admissible heuristic, its value for a state will always be lower or equal to the actual path cost. This can be explained intuitively by the fact that achieving all remaining preconditions usually needs additional actions. In

[4] This again does not technically conform to basic statement $B4$, since it only allows a union of size 2. We could form this union step by step, which would blow up the proof, but not exponentially.

the case where all other preconditions are achieved in parallel and are not deleted, we have $h^{max}(s) = h^*(s)$, which means that our h-value matches the perfect heuristic. In any case, our estimate can never be higher than the actual distance to the goal, because we will always need to achieve the most expensive precondition in order to reach the goal.

6.3.1 Proof of h-states for h^{max}

We have to prove that a non-expanded state s with $h^{max}(s) = h$ is an h-state. We can do so by using the knowledge we gained while calculating the h^{max} value of a state. We will use this knowledge in combination with the derivation rules from Definition 2 and the basic statements from Definition 3 in order to prove that s is an h-state.

Since we consider the variables which can be achieved from a state during the calculation of h^{max}, we will use these variables in order to define our x-state sets. We again use $g_s(v)$ as defined previously. In this case, s is the state with $h^{max}(s) = h$. For any variable $v \in s$, the value of $g_s(v)$ will be zero. For any variable $v \notin s$, the value is determined again by the cost of the most expensive precondition in addition to the action cost. We will build our set as follows. The set S_0 is again the set of all states S_{All}. Any layer x consists of all states, which contain only variables v for which $g_s(v) \leq h - x$ holds. We can reformulate this by stating that any layer x consists of all states which do not contain a variable v for which $g_s(v) > h - x$ holds. Hence, they contain $\neg v$. We define a set S_x as the set of states which contain $\neg v$ for any variable v for which $g_s(v) > h - x$ holds. Therefore, the state set S_1 contains only states which do not contain a variable v where $g_s(v)$ is larger than $h - 1$.

We know that $S_1, \dots, S_h$ cannot contain a goal, since at least one variable which is needed for the goal has not been achieved yet. This is because the h^{max} value is equal to the cost of the most expensive goal variable. We also know that successors of an x-state set can only be contained in an $(x-1)$-state set since they contain all states which are in S_x as well as all states which contain variables that can be reached with an additional action, which always has cost 1. Since $g_s(v) = 0$ for all variables in the state s, we know that s must be contained in the h-state set since $g_s(v) > h - h = 0$ for all variables $v \notin s$. We can now use these sets to prove that s is an h-state set.

(1) $D1 \rightarrow S_{All}$ is a 0-state set

(2) $B4 \rightarrow S_1[A] \subseteq S_{All}$

(3) $B3 \rightarrow S_1 \cap S_G \subseteq \emptyset$

(4) $D7, \{(1), (2), (3)\} \rightarrow S_1$ is a 1-state set

$\dots$

(n-2) $D7, \{$(n-6), (n-5), (n-4)$\} \rightarrow S_h$ is a h-state set

(n-1) $B1 \rightarrow \{s\} \subseteq S_h$

(n) $D3, \{$(n-2), (n-1)$\} \rightarrow \{s\}$ is an h-state set.

Using the same proof structure, we can show that $\{s\}$ is an h-state for any non-expanded state s in the task.

7 Conclusion

We have presented two possibilities to prove that the optimal solution of a task has been found. The first attempt made use of the already introduced certificates for unsolvability. In this approach, the original task was reformulated in such a way that it was allowed to be solved only with less than the initially determined cost, making the modified task unsolvable if the determined cost was optimal. Hence, an unsolvability certificate could be generated and verified for the task. In the second approach, an independent proof system for optimality was introduced. This system used the notion of x-state sets and was based on the proof system introduced for unsolvability. We will now discuss the results of the two approaches.

The first approach could be tested experimentally by modifying unit cost tasks as described. These modified tasks were then run using two different heuristics, namely $h^{M\&S}$ and h^{max}. In general, the results of the runs were good. Unfortunately, certificates could be generated for only slightly more than half of the runs. This was due to time and memory constraints used for the runs. However, most of the generated certificates could be verified. In addition, no severe memory or time overhead occurred for the successful runs. However, this approach is prone to error as the modification of the task might not be performed correctly, especially for tasks with special features. Since the translation of the tasks is never verified, these errors are not likely to be detected. Therefore, the second approach is intended to remedy the situation since it considers the original task without modifications.

In our second approach, we designed a formal proof system which used the notion of x-state sets, i.e. sets of states which have at least distance x to the goal. Using this notion, we could individually prove that a certain cost was indeed the optimal cost for the task by using the fact that the initial state had at least the corresponding distance to the goal. This proof system makes it possible to independently verify that the calculated cost of a task is indeed optimal.

Future work could include the extension of search algorithms by the generation of proofs based on the introduced proof system, as well as a stand-alone verifier for the proof system.

In order to actually implement these certificates, it is necessary to first investigate which types of state set representations are suitable for the certificates sketched.

This thesis only considered unit cost tasks. However, many tasks do not have the same cost for each action. It would therefore be interesting to extend the approaches presented for non-unit cost tasks.

Bibliography

[1] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129 (1):8–10, 2001.

[2] Salomé Eriksson, Gabriele Röger, and Malte Helmert. Unsolvability certificates for classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 88–97, 2017.

[3] Salomé Eriksson, Gabriele Röger, and Malte Helmert. A proof system for unsolvable planning tasks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 65–73, 2018.

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

[5] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[7] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 162–169, 2009.

[8] Richard Howey and Derek Long. VAL's progress: the automatic validation tool for PDDL2.1 used in the international planning competition. In *Proceedings of the International Conference on Automated Planning and Scheduling Workshop on the IPC*, pages 28–37, June 2003.

[9] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University, 1998.

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

A
Figures

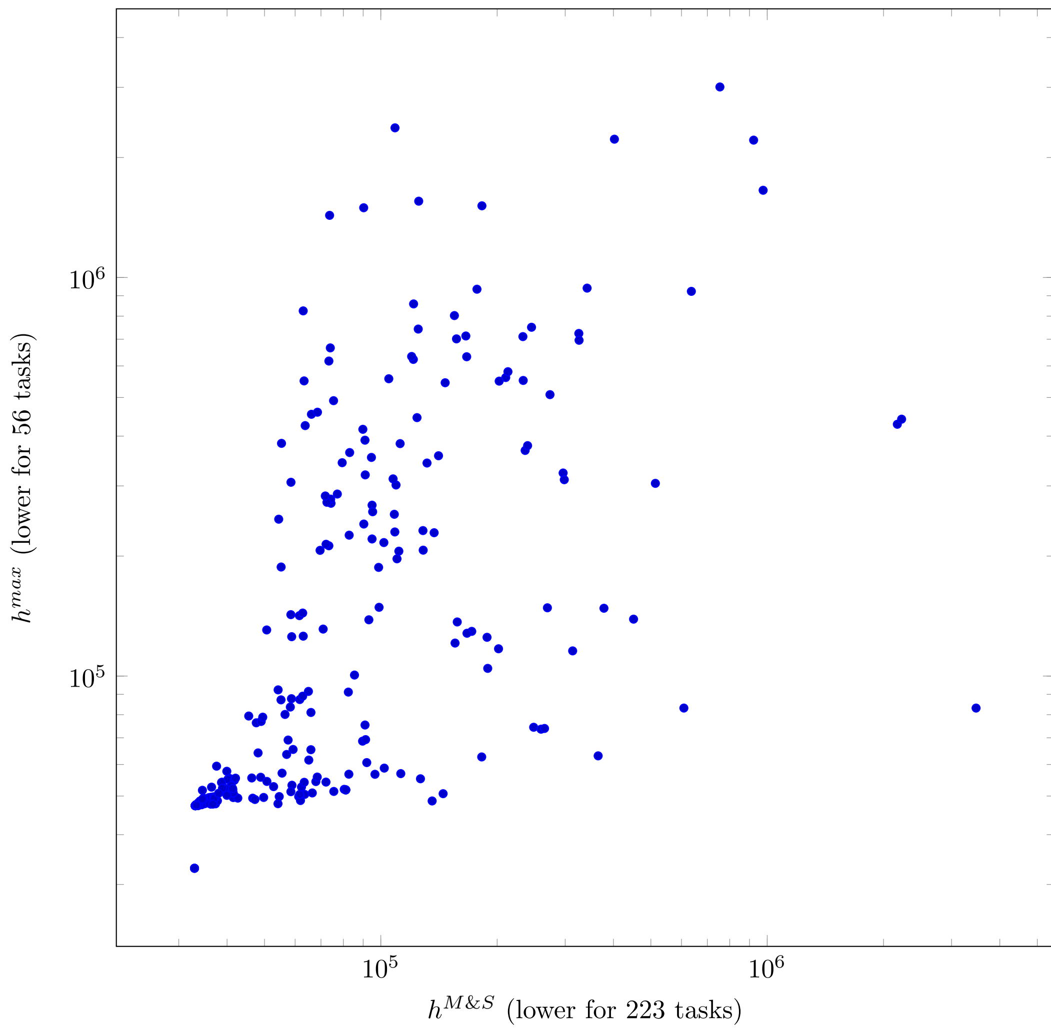

Figure A.1: Memory comparison between $h^{M\&S}$ and h^{max}

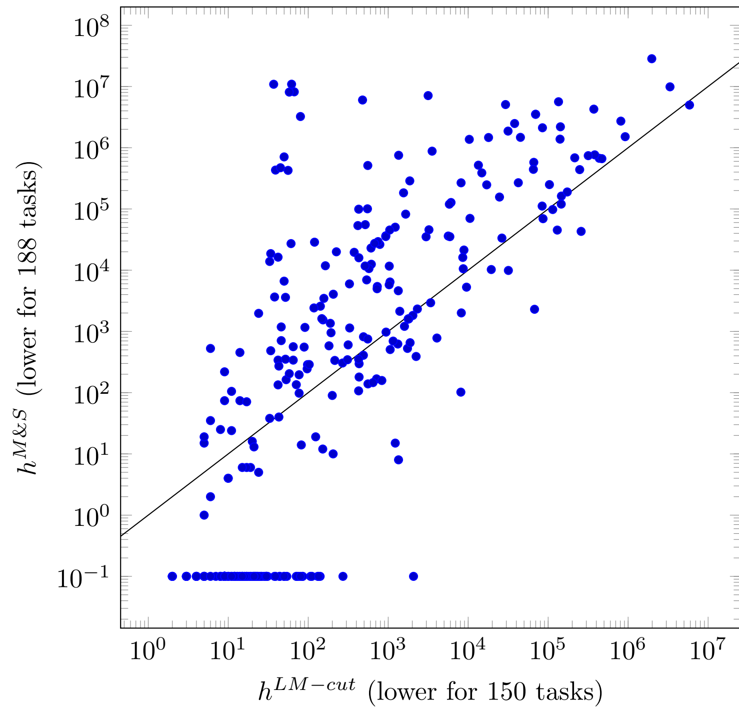

Figure A.2: Expansions comparison between h^{LM-cut} and $h^{M\&S}$

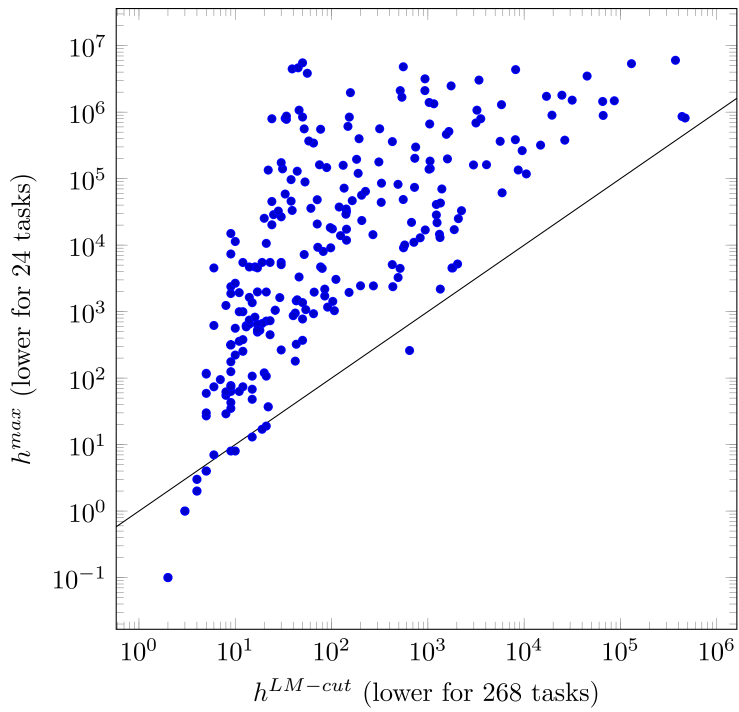

Figure A.3: Expansions comparison between h^{LM-cut} and h^{max}

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Esther Mugdan

Matriculation number — Matrikelnummer
2019-053-024

Title of work — Titel der Arbeit
Optimality Certificates for Classical Planning

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, 06.06.2022

Esther Mugdan

Signature — Unterschrift