

A Formal Verification of Strong Stubborn Set Based Pruning

Bachelor's thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence research group
ai.dmi.unibas.ch

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

Travis Rivera Petit
travis.riverapetit@stud.unibas.ch
2015-117-427

09.05.2020

Acknowledgments

I would like to thank very much my supervisor Dr. Salomé Eriksson for her thorough weekly feedback on my progress. I would also like to thank Dr. Florian Pommering for his help in clearing out some of my Isabelle/HOL questions.

Abstract

Classical Planning is a branch of artificial intelligence that studies single agent, static, deterministic, fully observable, discrete search problems. A common challenge in this field is the explosion of states to be considered when searching for the goal. One technique that has been developed to mitigate this is Strong Stubborn Set based pruning, where on each state expansion, the considered successors are restricted to Strong Stubborn Sets, which exploit the properties of independent operators to cut down the tree or graph search. We adopt the definitions of the theory of Strong Stubborn Sets from the SAS^+ setting to transition systems and validate a central theorem about the correctness of Strong Stubborn Set based pruning for transition systems in the interactive theorem prover Isabelle/HOL.

Table of Contents

Acknowledgments	ii
Abstract	iii
1 Introduction	1
1.1 Strong Stubborn Sets	2
1.2 Isabelle	2
2 Preliminaries	5
3 Strong Stubborn Sets for Transition Systems	8
4 Strong Stubborn Set based pruning	11
4.1 Proof of correctness in SAS^+	11
4.2 Proof of correctness in Transition Systems	14
5 Implementation	16
5.1 A Note on the pull function	17
6 Conclusion	19
Bibliography	20
Declaration on Scientific Integrity	21

1

Introduction

Classical Planning is one of the main studies of focus in AI research since it is a framework for modelling planning tasks that have sparked the interest of computer scientists since the inception of modern computers, ranging from solving Rubik's Cubes automatically to graph touring problems. It is also key in real world situations: consider the behavior of cranes and robot carts in the process of loading and unloading of cargo ships and moving containers around in a large harbour. What schedule of actions for the crane and the robots does the work with minimal cost? In essence, tackling this problem follows almost identical steps of reasoning as solving a Rubik's Cube in the minimum amount of steps does.

The theory of domain independent classical planning unifies the development of algorithms such that they may be used in any classical planning task with only minimal amount of work needed for coupling. This is achieved by abstracting the notion of a planning task and thinking of it as an mathematical object with discrete properties. The use of heuristic functions serves as a stand-in for domain-specific approaches that dictate which state is a good candidate to expand next.

A reoccurring problem in planning tasks is that their state spaces are too vast relative to our current processing power available. This makes it such that many algorithms with powerful theoretical properties such as A^* [6] or its variants become infeasible to use in practice. When confronted with a list of many potential states to explore, even under generous assumptions, the number of states to be expanded tends to grow exponentially on each iteration of a search algorithm [7]. State space pruning is a domain independent technique that narrows down that list while preserving optimality, thus reducing the computation overhead needed and making the use of some solvers feasible and more efficient.

A great deal of care must be taken when pruning a search space: if the the pruning method has errors it may convert a solvable task into an unsolvable one or remove all optimal solutions. However, due to the complexity of pruning algorithms, there have been published papers that defined pruning procedures with an incorrect behavior, even though they were –wrongly– proven to preserve optimality [4]. The difficulties arise because often there exist more than one optimal solution, and when confronted with two or more options, more than one of them could lead to an optimal solution, meaning that a pruning procedure may eliminate transitions that form optimal solutions as long as not all of them are removed.

Recently Mattmühler, Herlmert and Alkharaji obtained good experimental results with a Strong Stubborn Set based pruning technique [3]. They proved the correctness of their algorithm by using intuitive arguments, most of mathematics is done this way. However, due to the collective interest in rigorously proving the correctness of pruning procedures, a formal proof would set the reliability of the procedure in stone.

In this thesis we validate the proof of a central theorem on the correctness (in the sense of optimality preserving) of the Strong Stubborn Set pruning method in the interactive theorem prover Isabelle/HOL.

1.1 Strong Stubborn Sets

Often when confronted with a planning task one may find actions that commute. We consider as an example two a task of moving a bishop and a rook in a chessboard from the squares $a1$ and $d2$ to the squares $d2$ and $a1$ respectively while not threatening each other in any position in between, the initial position is shown in Figure 1.1. Here we consider the moves *bishop-a1-h8* and *rook-d2-g2*. Both are applicable in the starting position, and both are applicable after each other, moreover, the resulting position is the same no matter the order of moves. This observation, although quite evident, can be used to bypass a whole branch of a search tree: if *bishop-a1-h8* followed up by *rook-d2-g2* does not lead to an optimal solution, then neither will *rook-d2-g2* followed up by *bishop-a1-h8*. These two actions are an example of independent operators.

Things get particularly interesting when confronted with many independent operators: if op is an operator and π an array of operators such that for every operator op' in that path op' and op are independent, a lot of things can be said about π and its permutations on a given state depending on the behavior of op on that state, cutting down the search exponentially. Strong Stubborn Sets were conceived with the idea of exploiting facts about independent operators to narrow down tree or graph searches, for instance if op is in a Strong Stubborn Set and is applicable in a state s , then only the operators it is dependent of must also belong to that set, although further conditions must also be met to ensure the desired behaviour when pruning.

Stubborn sets have been first proposed in the area of model checking [5] and have since been adapted to SAS^+ planning tasks [3]. In Chapter 3 of this thesis we show how they can be adapted to the more general setting of transition systems.

1.2 Isabelle

Isabelle is an interactive computer program where human and machine team up to try to prove mathematical formulas in a logical calculus together. It also facilitates the use of external automatic theorem provers and supports functional programming. The software is open source, and a mirror of the source code is available on GitHub [2]. Isabelle has been originally developed by the Technische Universität München and the University Of Cambridge, but has received contributions from many institutions and individuals since.

A challenge when doing mathematics at a high level of granularity, and even in mathematics

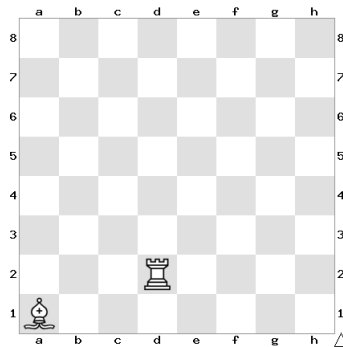


Figure 1.1: The initial position of a planning task where the goal is to move the rook to $a1$ and the bishop to $d2$ without having the pieces threatening each other in any position.

in general is that we are often tricked by our common sense into thinking that certain claims are so evident and elementary that their proof follows must immediately by the definitions of the objects involved. And even when we know they do not, we tend to use meta-mathematical reasoning to argue their validity. This is potentially dangerous as it opens up the possibility of taking things for granted that are false and ending up with an incorrect proof, often for a false claim. Isabelle offers an alternate way of doing mathematics where every logical step has to be formally justified, and if such justification is not sound, Isabelle lets the user know in real time. Figure 1.2 shows a screenshot of Isabelle's default IDE Isabelle/jEdit, and Figure 1.3 shows an instance where Isabelle cannot conclude a statement from a premise. Working with Isabelle has the benefit of forcing the user into being rigorous and of giving its proofs a soundness stamp. For instance if the P vs NP problem were to be proven in Isabelle, we would have a high level of certainty that the proof is correct.

A notable example of an Isabelle proof includes the correctness of an operating system's kernel, where the Isabelle proof spans roughly two hundred thousand lines [8].

As of the time of this writing, Isabelle's archive of formal proofs [1] contains 530 articles of formal proofs in mathematics and computer science written in Isabelle.

Isabelle/HOL is an instance of Isabelle for proving theorems in higher order logic which is equipped a high level of expressiveness, for instance recursive functions with pattern matching can be defined as long as it is proven that such functions are well defined. This is especially useful when trying to model and prove facts about algorithms that are recursive in nature with complex pattern matchings.

```

1186 from this and sInt and lem4 and <optimal_plan T s pi ^ pi ≠ []> have validOperatorsInPi: "∀ op ∈ set pi. valid_operator T op" b
1187
1188 (* obtain disjunctive landmark L *)
1189 from strong_stubborn_set_def and strStbSet have "∃ L: 'a operator set, disjunctive_landmark T L s ^ L ⊆ SSS" by fastforce
1190 from this obtain L :: "'a operator set" where "disjunctive_landmark T L s ^ L ⊆ SSS" by auto
1191
1192 (* obtain a valid operator that is in both SSS and pi that is the first occurrence of pi and is also in SSS *)
1193 from this and validTs and <optimal_plan T s pi ^ pi ≠ []> and lem1 have interNotEmpty: "SSS ∩ (set pi) ≠ {}" by fastforce
1194 from this have "∃ op ∈ SSS ∩ set pi. op ≠ {}" using valid_operator_def by auto
1195 from lem2 and interNotEmpty and this have "first_occurrence pi SSS ∩ set pi" using <optimal_plan T s pi ^ pi ≠ []> by bla
1196 from this and validOperatorsInIntersection obtain op :: "'a operator" where "op = first_occurrence pi SSS ^ valid_operator T op ^
1197
1198 (* op cannot be s.t. it is not applicable in s *)
1199 from this have opInPi: "op ∈ set pi" by auto
1200 from <disjunctive_landmark T L s ^ L ⊆ SSS> and strStbSet and <op = first_occurrence pi SSS ^ valid_operator T op ^ op ∈ SSS ∩ s
1201 from this have "¬ applicable op s ⇒ ∃ op' ∈ SSS. comes_before op' op pi ^ op' ∈ set pi" by (meson <goal ∈ goals T ^ is_path_fr
1202 from this have "¬ applicable op s ⇒ ∃ op' ∈ SSS. op ∈ SSS ^ op' ∈ set pi ^ op' ∈ set pi ^ valid_operator T op ^ valid_operator
1203 from this and lem6 have "¬ applicable op s ⇒ False" by auto
1204
1205 (* thus op is applicable in s *)
1206 from this have applicable: "applicable op s" by auto
1207
1208 (* op is cannot be s.t. it is dependent from any of its previous operators in pi *)
1209 from validTs and strStbSet and sInt and <optimal_plan T s pi ^ pi ≠ []> and <op = first_occurrence pi SSS ^ valid_operator T op ^
1210
1211 (* thus op is independent from the operators that come before it *)
1212 from this have "op' ∈ set pi ^ comes_before op' op pi ⇒ independent T op op'" by auto
1213
1214 (* independent operators commute and preserve optimality, as long as the last one is applicable in s *)

```

```

proof (chain)
  picking this:
    • ¬ applicable op s ⇒
      ∃op' ∈ SSS
      op' ∈ SSS ^
      op' ∈ set pi ^
      op' ∈ set pi ^ valid_operator T op ^ valid_operator T op' ^ op = first_occurrence pi SSS ^ comes_before op' op pi
    • ?op' ∈ ?SSS ⇒
      ?op' ∈ ?SSS ⇒ ?op' ∈ set ?pi ⇒ ?op' ∈ set ?pi ⇒ ?op' = first_occurrence ?pi ?SSS ⇒ comes_before ?op' ?op ?pi ⇒ False

```

Figure 1.2: Isabelle’s IDE Isabelle/jEdit. It provides immediate feedback on the status of proofs, syntax highlighting and T_EX -like semantic markup.

```

859 then show ?case using Nil.preds(1) Nil.preds(2) Nil.preds(5) by auto
860 next
861 case (Cons x xs)
862 {
863   assume "xs = []"
864   from this have "s ∈ states T ⇒ applicable op s ⇒ op ≠ x ⇒ path_independence T [x] op ⇒ plan T s ((x, op) @ ys) ⇒ plan T s (swap_with_previous ((x, op)
865   from this have "s ∈ states T ⇒ validTs T ⇒ applicable op s ⇒ op ≠ x ⇒ independent T x op ⇒ path_independence T x op ⇒ plan T s ((x, op) @ ys) ⇒ plan T s ((op, x) @ ys) ⇒ ?c
866   from this have ?case using Cons.preds(1) Cons.preds(2) Cons.preds(4) Cons.preds(5) <<xs = []> path_independence.simps(2) plan_15 validTs by force
867 }
868 }
869 moreover {
870   assume "xs ≠ []"
871   from this have "s ∈ states T ⇒ applicable op s ⇒ op ≠ x ⇒ path_independence T (x#xs) op ⇒ plan T s ((x#xs) @ [op] @ ys) ⇒ plan T s (swap_with_previous
872   from this have "s ∈ states T ⇒ applicable op s ⇒ op ≠ x ⇒ independent T op x ⇒ path_independence T xs op ⇒ plan T s ((x#xs) @ [op] @ ys) ⇒ plan T s (x
873   from this have "s ∈ states T ⇒ applicable op s ⇒ op ≠ x ⇒ path_independence T xs op ⇒ applicable x s ⇒ plan T (effect T x s) (x
874   from this have "s ∈ states T ⇒ applicable op s ⇒ op ≠ x ⇒ independent T op x ⇒ path_independence T xs op ⇒ applicable x s ⇒ plan T (effect T x s) (x
875   from this have ?case
876     by (metis Cons.hyps Cons.preds(3) Cons.preds(4) Cons.preds(5) append_Cons enables_def independent_def independent_symmetric is_path_from_to_simp
877   }
878 ultimately show ?case by auto
879 qed

```

```

Failed to finish proofs:
goal (1 subgoal):
1. (xs = []) ⇒
  plan T s
  (if x = op then x # op # ys
  else if op = op then op # x # ys
  else x # swap_with_previous (op # ys) op) ⇒
  (xs ≠ [] ⇒
  plan T s (swap_with_previous (x # xs @ op # ys) op) ⇒
  plan T s (swap_with_previous (x # xs @ op # ys) op) ⇒
  plan T s (swap_with_previous ((?x2 # ?xs2) @ [op] @ ys)) ⇒ plan T ?sa2 (swap_with_previous ((?x2 # ?xs2) @ [op] @ ys) op)

```

Figure 1.3: An example of Isabelle/jEdit’s signaling of a proof gone wrong, where Isabelle’s proof method `auto` cannot conclude the claim `?case` from the assumptions in the lines 859 and 865.

2

Preliminaries

We build up the definitions needed for the theory of Strong Stubborn sets. We do this by doing some adaptations to the definitions from [3]. These changes make talking about operators, planning tasks and states easier in subsequent chapters but do not change the semantics of the content.

We start by building up the theory of SAS^+ planning tasks.

Definition 1. *An alphabet is a finite non-empty set.*

In our setting, we distinguish between two types of alphabets: variable alphabets, which contain names of variables, and domain alphabets, which contain the values that variables may take. For instance, in the game of chess, a variable alphabet would be the algebraic names of the square coordinates $\{a, \dots, h\} \times \{1, \dots, 8\}$ and a domain alphabet would contain the chess pieces that might occupy these squares, namely $(\{\text{white}, \text{black}\} \times \{\text{king}, \text{queen}, \text{rook}, \text{bishop}, \text{knight}, \text{pawn}\}) \cup \{\text{empty}\}$.

Definition 2. *Let V be a variable alphabet and D be a domain alphabet, a global domain is a mapping $d : V \rightarrow \mathcal{P}(D)$ ¹ i.e. each variable v gets assigned to a variable domain $d(v)$.*

Definition 3. *Let V be a variable alphabet, D be a domain alphabet and $d : V \rightarrow \mathcal{P}(D)$ be a global domain. A state s is a mapping from V to D where for each $v \in V$ it holds that $s(v) \in d(v)$. A partial state is a state where the mapping $V \rightarrow D$ is partial.*

A state defines the current configuration of a planning task via a function, the initial state from Figure 1.1 is a function that maps the $a1$ square to the bishop piece, the $d2$ square to the rook piece, and the remaining 62 squares to *empty*.

Definition 4. *An operator op is a triple $op = \langle pre(op), cost, eff(op) \rangle$ where $cost \in \mathbb{R}_{\geq 0}$ and $pre(op)$ and $eff(op)$ are partial states called the precondition and effect of op respectively.*

With this machinery we now can define planning tasks:

¹ By $\mathcal{P}(D)$ we mean the power set of D .

Definition 5. A SAS⁺ planning task Π is a 6-tuple

$$\Pi = \langle V, D, d, s_0, s_*, O \rangle$$

where V is a variable alphabet, D is a domain alphabet, $d : V \rightarrow D$ is a global domain, s_0 is a state (called initial state) and s_* is a partial state (called goal). Finally, O is a set of operators.

Coming back to the chess puzzle from Figure 1.1 we may let Π be the planning task whose set of operators O contains all of the legal moves for the bishop and rook via preconditions and effects, for instance for $bishop-a1-h8 \in O$, its precondition may be defined as follows

$$\begin{aligned} (a, 1) &\mapsto \text{bishop} \\ (i, j) &\mapsto \text{empty} \quad \forall (i, j) \in \{(b, 2), (c, 3), \dots, (h, 8)\} \\ (i, 8) &\mapsto \text{empty} \quad \forall i \in \{a, \dots, g\} \\ (h, i) &\mapsto \text{empty} \quad \forall i \in \{1, \dots, 7\} \end{aligned}$$

While its effect would then be a partial state with only two mappings defined: $(a, 1) \mapsto \text{empty}$ and $(h, 8) \mapsto \text{bishop}$.

We are still missing the notion of a solution, intuitively it is a sequence of operators that ‘lead’ one state to the goal. We build up the definitions needed for solutions now.

Definition 6. Let s be a state and s_p be a partial state. s_p agrees with s if for all variables v where s_p is defined, $s(v) = s_p(v)$.

Definition 7. Let op be an operator and s be a state. op is applicable in s if the precondition of op agrees with s .

Definition 8. Let s be a state, op be an operator applicable in s and s_p be the effect of op . We let $s[op]$ be the state defined as follows:

$$s[op](v) = \begin{cases} s(v) & \text{if } s_p(v) \text{ is undefined} \\ s_p(v) & \text{otherwise} \end{cases} \quad \forall v \in V.$$

We extend this definition as follows: Let $n \in \mathbb{N}$. Then

$$s[op_1, \dots, op_n](v) := \begin{cases} s[op_1, \dots, op_{n-1}](v) & \text{if } \text{eff}(op_n)(v) \text{ is undefined} \\ \text{eff}(op_n)(v) & \text{otherwise} \end{cases}$$

assuming applicability, namely that op_i is applicable in $s[op_1, \dots, op_{i-1}]$ for $i = 2, \dots, n$ and that op_1 is applicable in s .

The previous definition is a natural way of defining chains of operators: $s[op_1, \dots, op_n]$ is the state that is obtained after applying op_1, \dots, op_n (in that order) to s .

Definition 9. A path π from a state s to a state s' is a finite sequence of operators op_1, \dots, op_n such that op_1 is applicable in s , op_i is applicable in $s[op_1, \dots, op_{i-1}]$ for $i = 2, \dots, n$ and $s[op_1, \dots, op_n] = s'$.

We let $\text{cost}(\pi)$ be the sum of the costs of all the operators in π .

Definition 10. Let $\Pi = \langle V, D, d, s_0, s_*, O \rangle$ be a SAS⁺ planning task, a solution for a state s is a path whose elements lie in O from s to a state s' where s_* agrees with s' . An optimal solution π for s is a solution satisfying

$$\pi = \underset{\text{solution } \pi' \text{ for } s}{\operatorname{arg\,min}} \quad \operatorname{cost}(\pi')$$

Note that, a priori, nothing can be said about the existence of a solution (and for that matter, of an optimal solution). A planning task may have no solution, one or many solutions, or an infinite number of solutions. Also, when we talk about the solution of a planning task, we refer to a solution for the initial state s_0 .

Moving on, we build up the definitions necessary for the theory of Strong Stubborn Sets.

Definition 11. Let $\Pi = \langle V, D, d, s_0, s_*, O \rangle$ be a planning task and $op, op' \in O$. We say that

- op disables op' if there exists a $v \in V$ such that $\operatorname{eff}(op)(v)$ and $\operatorname{pre}(op')(v)$ exist but are unequal.
- op and op' conflict if there exists a $v \in V$ such that $\operatorname{eff}(op)(v)$ and $\operatorname{eff}(op')(v)$ exist but are unequal.
- op and op' are dependent if at least one of them disables the other or they conflict. We write $\operatorname{dep}(op)$ as a stand-in for the set of all the operators in O that are dependent of op .

Definition 12. Let s be a state and s_p be a partial state. A disjunctive action landmark for s_p in s is a set of operators L such that for every path π from s to a state that s_p agrees with, π contains at least one operator from L .

Definition 13. Let s be a state and op be an operator not applicable in s . A necessary enabling set for an operator op in state s is a disjunctive action landmark for $\operatorname{pre}(op)$ in s .

Necessary enabling sets have their name because if op is an operator not applicable in a state s and π is some path from s to a state on which op is applicable, then a necessary enabling set for op in s contains some operator in π that is necessary, although generally not sufficient, to actively enable op .

Again, taking the chess puzzle from 1.1 but restricting the grid to the dimensions of 4 by 3, i.e where the corners of the board are $(a, 1), (a, 3), (d, 1), (d, 3)$, a necessary enabling set for the operator *bishop-b2-c1* in the initial position is $\{\operatorname{rook-d2-d3}\}$ or any of its supersets.

Definition 14. Let $\Pi = \langle V, D, d, s_0, s_*, O \rangle$ be a planning task and $s \in V$. SSS is a Strong Stubborn Set in s if it contains a disjunctive action landmark for s_* in s and if for all $op \in \operatorname{SSS}$

- if op is applicable in s , then $\operatorname{dep}(op) \subseteq \operatorname{SSS}$
- if op is not applicable in s , then SSS contains some necessary enabling set of op in s .

3

Strong Stubborn Sets for Transition Systems

We define transition systems and adopt the theory of Strong Stubborn Sets to this new setting.

In the previous chapter, we have seen that states were a particular kind of mapping where variables got assigned to values of their corresponding domains. Transition systems simplify this: in this framework states are now atomic units. In a way, transition systems behave like weighted directed graphs, where states can be thought of as vertices and actions as collections of edges.

Definition 15. A Transition System \mathcal{T} is a 6-tuple

$$\mathcal{T} = \langle S, T, A, cost, s_0, G \rangle$$

where S is a finite set of states, $T \subseteq S \times A \times S$ is a set of transitions, A is a set of actions, $cost$ is a function $A \rightarrow \mathbb{N}_0$, $s_0 \in S$ is the initial state and $G \subseteq S$ is the set of goal states.

Where in SAS^+ planning tasks we had a set of variables and a global domain, here we only have a set of states. Actions can be thought of as operator names/identifiers, and operators themselves are sets of transitions meeting certain conditions.

We note that transition systems are more general than SAS^+ planning tasks, for instance, they allow for more flexibility with goal states. Coming back to the chess example, the goal *rook-at-a1-and-bishop-at-d2* can be defined with a partial state in SAS^+ that maps the *a1* square to the rook piece, the *d2* square to the bishop piece, and is undefined for all the other variables, but a goal of the form *rook-at-a1-and-bishop-at-d2-OR- rook-at-a8-and-bishop-at-c7* cannot be expressed via a partial state. In Transition systems, however such a goal can be dealt with easily by letting G be the set of all the states where the rook and the bishop are in one of the two configurations mentioned earlier.

For a transition $t = \langle s, a, s' \rangle \in T$ we let $src(t) = s$, $dst(t) = s'$ and $act(t) = a$ and call a Transition System \mathcal{T} valid if $\forall t, t' \in T : (src(t) = src(t') \wedge act(t) = act(t')) \longrightarrow t = t'$, that is, a Transition System is valid if actions are not ambiguous in the sense that actions may not map states to more than one successor state.

From now on we will also let $\mathcal{T} = \langle S, T, A, cost, s_0, G \rangle$ be an arbitrary valid transition system.

We move on by adopting our own definitions for the theory of Strong Stubborn Sets in transition systems. The definitions are partly adopted from the previous chapter and [9], which is itself an adaptation of [5].

Definition 16. *An operator op in \mathcal{T} is a non-empty subset of T such that $\forall t, t' \in op : act(t) = act(t')$.*

When the context makes it clear, we use the convention of calling op an operator instead of an operator in \mathcal{T} .

Definition 17. *An operator op is applicable in $s \in S$ iff there exists a transition t in op such that $src(t) = s$.*

If op is applicable in s , we write $app(op, s)$ for short.

Note that in a valid transition system \mathcal{T} , for any $s \in S$ and any operator op in \mathcal{T} there is at most one $t \in op$ such that $src(t) = s$. Applicability restricts this further: it states that there is at least one t with such property. Thus if op is applicable in s and if \mathcal{T} is valid then there exists exactly one $t \in T$ such that $s = src(t)$.

This motivates the following definition.

Definition 18. *For an operator op that is applicable in $s \in S$ in a valid Transition System \mathcal{T} we let $effect(op, s)$ be the state $s' \in S$ such that $\exists t \in op : src(t) = s \wedge dst(t) = s'$.*

We continue with the definitions leading up to Strong Stubborn Sets.

Definition 19. *Let op and op' be operators.*

1. *op enables² op' if for every $s \in S$, if op and op' are applicable in s , then op' is applicable in $effect(op, s)$.*
2. *op and op' are non conflicting if*

$$app(op, s) \wedge app(op', s) \wedge app(op, effect(op', s)) \wedge app(op', effect(op, s))$$

implies

$$effect(op', effect(op, s)) = effect(op, effect(op', s))$$

for every $s \in S$.

3. *op and op' are independent if they enable each other and they are non conflicting.*

The proof of Lemma 1 in Chapter 4 provides a strong intuition as for why this definition of enables is compatible with the negation of disables in Definition 11. The same applies to this definition of non-conflicting.

The definition of path, path effect and solution can be defined in an analogous way as has been done in the previous chapter.

² We use enables to mean the negation of disables, which is not to be confused with active enabling.

Definition 20. Let op be an operator not applicable in $s \in S$. N is a necessary enabling set for op in s if for every solution π for s the following holds: if op is a member of π , then there exists some op' such that op' comes before op in π and $op' \in N$.

There is a slight semantic nuance between the Definitions 20 and 12: the latter allows for the definition to hold for paths between any two states. In this thesis we only consider necessary enabling sets from states to goals, which is reflected in Definition 20.

Definition 21. A disjunctive action landmark L for a state $s \in S$ is a set of operators such that for every solution for s , there exists an operator in that path that is also in L .

Definition 22. A Strong Stubborn Set SSS for $s \in S$ if the following hold:

- SSS contains a disjunctive action landmark for s .
- if $op \in SSS$ and $\neg app(op, s)$ then SSS contains a necessary enabling set for op in s .
- if $op \in SSS$ and $app(op, s)$ then SSS contains all the operators op' for which op and op' are dependent.

4

Strong Stubborn Set based pruning

On each iteration of a search algorithm such as A^* , a pruning technique creates a blacklist of elements that should not be expanded because they either do not lead to a goal state, or do not start an optimal solution for the given state, or do start an optimal solution but can be safely ignored because there is at least one such equivalent element that is not pruned. A central theorem in the theory of Strong Stubborn Sets shows that in an active state³ s , any Strong Stubborn Set for s includes an operator that starts an optimal solution. This theorem is the main focus of this thesis, it stated formally as follows:

Theorem 1. *Let $\Pi = \langle V, D, d, s_0, s_*, O \rangle$ be a SAS^+ planning task, s be active in Π and SSS be a Strong Stubborn Set for s . Then there exists an $op \in SSS$ that starts some optimal solution for s .*

In essence, this means that restricting expansion to elements of Strong Stubborn Sets preserves optimality.

There is a lot to be said about which Strong Stubborn Sets should be chosen when expanding, computing ‘some’ Strong Stubborn Set when choosing which state to expand next is a non-deterministic procedure. It is easy to see that the set of all applicable operators in a state is indeed a Strong Stubborn Set, albeit a not useful one, since taking that route would prune zero operators every time. On the other extreme, finding Strong Stubborn Sets of minimal size is not feasible, so the computation overhead for finding such sets is too inefficient for it be of use in practice.

4.1 Proof of correctness in SAS^+

We provide the proof of Theorem 1 from [3] elaborating further on their arguments. We fix a planning task Π . For a path $\pi = \langle op_1, \dots, op_n \rangle$ we let $swp(\pi, op)$ be the function that swaps the first occurrence op in with the element preceding it, namely $swp(op_i, \pi) = \langle op_1, \dots, op_i, op_{i-1}, op_{i+1}, \dots, op_n \rangle$ (granted that op_1, \dots, op_{i-1} are all unequal to op), and if op_i is the first element of π , then $swp(op_i, \pi) = \pi$. We first prove two important results:

³ Active states are those for which there exists a path to the goal.

Lemma 1. *Let π be an optimal solution for a state s in Π and let op be an element in π that is applicable in s such that it is independent from all of the operators that come before it. Then $swp(op, \pi)$ is an optimal solution for s .*

Proof. Formally, we must show that

$$\pi \text{ is an optimal solution for some state } s \implies swp(\pi, op) \text{ is an optimal solution for } s$$

assuming that op is in π , all of the elements of π that come before op are independent of op and op is applicable in s .

We let \cdot be the function that concatenates two sequences together and use it with an infix notation, i.e. $\langle x_1, \dots, x_m \rangle \cdot \langle y_1, \dots, y_n \rangle := \langle x_1, \dots, x_m, y_1, \dots, y_n \rangle$. We first note that if π is a path containing op , then $\pi = \mathbf{xs} \cdot \langle op \rangle \cdot \mathbf{ys}$ for some sequences \mathbf{xs} and \mathbf{ys} where for every element x in \mathbf{xs} : $x \neq op$. We prove the result over induction on length of \mathbf{xs} .

The first base case is when \mathbf{xs} is the empty sequence. Then op is the first element of π so $swp(\pi, op) = \pi$ and we are done.

For our second base case we consider \mathbf{xs} having length one: $\mathbf{xs} = [op']$ for some operator op' . We assume the premise and let s be a state for which π is an optimal solution. It suffices to show that op is applicable in $s[op']$ and that $s[op, op'] = s[op', op]$ because then $s[\pi] = s[swp(\pi, op)]$, and since $cost(\pi) = cost(swp(\pi, op))$, $swp(\pi, op)$ would then also be an optimal solution. By assumption op and op' are not dependent, in particular op enables (that is, not disables) op' . By the definition of disables:

$$\begin{aligned} \neg(op \text{ disables } op') &\iff \\ \neg(\exists v \in V : eff(op)(v) \text{ exists} \wedge pre(op')(v) \text{ exists} \wedge eff(op)(v) \neq pre(op')(v)) &\iff \\ \forall v \in V : (eff(op)(v) \text{ exists} \wedge pre(op')(v) \text{ exists}) \implies eff(op)(v) = pre(op')(v) & \end{aligned}$$

In particular for s , since op is applicable, it follows that

$$\forall v \in V : pre(op)(v) \text{ exists} \implies s[op](v) = pre(op')(v)$$

So $pre(op')$ and $s[op]$ agree, thus op' is applicable in $s[op]$. Using the same reasoning we can also see that op' is applicable in $s[op]$.

By Definition 8 we have

$$s[op, op'](v) = \begin{cases} s(v) & \text{if } eff(op')(v) \text{ is undefined and } eff(op)(v) \text{ is undefined} \\ eff(op)(v) & \text{if } eff(op')(v) \text{ is undefined and } eff(op)(v) \text{ is defined} \\ eff(op')(v) & \text{if } eff(op')(v) \text{ is defined} \end{cases}$$

and similarly

$$s[op', op](v) = \begin{cases} s(v) & \text{if } eff(op)(v) \text{ is undefined and } eff(op')(v) \text{ is undefined} \\ eff(op')(v) & \text{if } eff(op)(v) \text{ is undefined and } eff(op')(v) \text{ is defined} \\ eff(op)(v) & \text{if } eff(op)(v) \text{ is defined} \end{cases}$$

By pattern matching we can see that if v is such that at most one of $\text{eff}(op)$ and $\text{eff}(op')$ is undefined for v , then $s[op, op'](v) = s[op', op](v)$. If v is defined for both, then by the definition of conflicting:

$$op \text{ does not conflict with } op' \iff$$

$$\neg(\exists v \in V : \text{eff}(op)(v) \text{ exists} \wedge \text{eff}(op')(v) \text{ exists} \wedge \text{eff}(op)(v) \neq \text{eff}(op')(v)) \iff \\ \forall v \in V : (\text{eff}(op)(v) \text{ exists} \wedge \text{eff}(op')(v) \text{ exists}) \longrightarrow \text{eff}(op)(v) = \text{eff}(op')(v)$$

So $s[op, op'](v) = s[op', op](v)$, finishing the proof of this base case.

For the inductive step we show that if

$$\pi \text{ is an optimal solution for some state } s \implies \text{swp}(\pi, op) \text{ is an optimal solution for } s$$

assuming that op is in π , all of the elements of π that come before op are independent of op and op is applicable in s , then

$$\langle x \rangle \cdot \pi \text{ is an optimal solution for some state } s' \implies \text{swp}(\langle x \rangle \cdot \pi, op) \text{ is an optimal solution for } s'$$

assuming that op is in $\langle x \rangle \cdot \pi$, all of the elements of $\langle x \rangle \cdot \pi$ that come before op are independent of op and op is applicable in s' .

We assume the premise and let s be a state such that π and $\text{swp}(\pi, op)$ are optimal solutions for s and write π as $\langle x_1, \dots, x_N \rangle \cdot \langle op \rangle \cdot \mathbf{ys}$ where $x_1 \neq op, \dots, x_N \neq op$.

If $N = 0$ the claim holds by the second base case. If $N \geq 1$ and $x = op$ the claim holds by the first base case. If $N \geq 1$ and $x \neq op$ we derive

$$\text{swp}(\langle x \rangle \cdot \langle x_1, \dots, x_N \rangle \cdot \langle op \rangle \cdot \mathbf{ys}, op) = \text{swp}(\langle x, x_1, \dots, x_N, op \rangle \cdot \mathbf{ys}, op) \\ \stackrel{N \geq 1}{=} \langle x, x_1, \dots, x_{N-1}, op, x_N \rangle \cdot \mathbf{ys} = \langle x \rangle \cdot \langle x_1, \dots, x_{N-1}, op, x_N \rangle \cdot \mathbf{ys} \\ = \langle x \rangle \cdot \text{swp}(\langle x_1, \dots, x_N \rangle \cdot \langle op \rangle \cdot \mathbf{ys}, op) = \langle x \rangle \cdot \text{swp}(\pi, op)$$

Furthermore, if $\langle x \rangle \cdot \pi$ is an optimal solution for s , then π is an optimal solution for $s[x]$. Thus we must show that

$$\pi \text{ is an optimal solution for } s[x] \implies \text{swp}(\pi, op) \text{ is an optimal solution for } s[x]$$

The proof follows directly by letting $s' = s[x]$. □

Corollary 1. *Let π be an optimal solution for an $s \in S$ and let op be an element in π applicable in s such that it is independent from all of the operators that come before it. Then op starts an optimal solution for s .*

Proof. Let n be the length of π . Use Lemma 1 $n - 1$ times on π and s . □

With this out of the way we are ready to prove Theorem 1.

Proof. Let op_1, \dots, op_n be the members of an optimal solution π . Since SSS is a disjunctive action landmark for s_* , $\{op_1, \dots, op_n\} \cap SSS \neq \emptyset$. Let then op be the member with the smallest index of π that is also in an element of SSS .

Assume op is not applicable on s , then by the definition of Strong Stubborn Sets and disjunctive action landmarks, there would exist at least one operator that comes before op in π that is also contained in SSS . This is a contradiction since op is the element that is contained in both π and SSS with the smallest index, hence op is applicable in s .

Assume there exists an op' that comes before op in π op such that op and op' are dependent and fix it. By the definition of Strong Stubborn Sets $op' \in SSS$. Then all of the following hold:

- $op' \in SSS$
- op' in π
- op' comes before op in π

Which contradicts the fact that op is the element with the smallest index in π that is also in SSS , so op is independent from all of the operators that come before it in π . We use Corollary 1 on π and s , the proof follows directly. \square

4.2 Proof of correctness in Transition Systems

We prove an analog of Theorem 1 for transition systems. We fix a valid Transition System $\mathcal{T} = \langle S, T, A, cost, s_0, G \rangle$.

Theorem 2. *Let $s \in S$ be an active state and SSS be a Strong Stubborn Set for s . Then there exists an $op \in SSS$ that starts some optimal solution for s .*

Like before, we rely heavily on a corollary of an analog of Lemma 1:

Lemma 2. *Let π be an optimal solution for a $s \in S$ and let op be an element in π applicable in s such that it is independent from all of the operators that come before it. Then $swp(\pi, op)$ is an optimal solution for s .*

Here swp is defined as in Lemma 1.

Proof. (of Lemma 2). We write π as $\mathbf{xs} \cdot \langle op \rangle \cdot \mathbf{ys}$ and prove the claim by induction on \mathbf{xs} . For the first base case we consider the \mathbf{xs} as the empty sequence. Then the first element of π is op and $swp(\pi, op) = \pi$ so $s[\pi] = s[swp(\pi, op)]$ and we are done.

For our second base case we let $\mathbf{xs} = [op']$ for some operator op' in \mathcal{T} that is independent of op . By the definition of independence and enables, op is applicable in $effect(op, s)$ and op' is applicable in $effect(op, s)$, and then by the definition of conflict, $effect(op, effect(op', s)) = effect(op', effect(op, s))$ and we are done.

The inductive step is proved the same way as it has been done in Lemma 1. \square

Corollary 2. *Let π be an optimal solution for a $s \in S$ and let op be an element in π applicable in s such that it is independent from all of the operators that come before it. op starts some optimal solution for s .*

Proof. Let n be the length of the path π . Use Lemma 2 $n - 1$ times on π and s . \square

With this we prove Theorem 2.

Proof. SSS contains a disjunctive action landmark for s , so there exists at least one element in π that is also contained in SSS . Let then op be the element of the smallest index in π that is also an element of SSS .

Assume op is not applicable in s , then by Definition 22, SSS is a necessary enabling set for op in s , thus $\exists op' : op'$ comes before op in $\pi \wedge op' \in SSS$ which leads to a contradiction because otherwise op would no longer be the element with the smallest index in π that is also in SSS . Hence $app(op, s)$.

Furthermore, assume there exists an op' in π that comes before op and is dependent of op . By Definition 22, given that op is applicable in s , $op' \in SSS$, again leading to a contradiction. By Corollary 2, there exists an $op \in SSS$ that starts an optimal solution for s . \square

5

Implementation

In Isabelle/HOL, the proof of Theorem 2 followed a bottom up approach by starting from elementary lemmas and slowly building up. While our proof of the theorem in Chapter 4 has been achieved with a few paragraphs of arguments, the formal proof takes roughly one thousand lines. As is often the case in discrete mathematics, most of the claims have been proven by induction.

For the proof we considered transition systems as directed weighted graphs, and operators were defined separately. We did this because reasoning about arbitrary transition systems \mathcal{T} requires to consider every single possible operator for \mathcal{T} . Because of this, we defined operators more broadly, independent of transition systems as sets of transitions, and defined a predicate `valid_operator` that filters out operators that are not well defined for \mathcal{T} .

This had the benefit of working with simpler data structures at the price of having to keep track of multiple instances of mathematical objects at the time. It also made it possible for functions to be defined over sets of operators, which simplified the syntax. However, because operators have been defined so broadly, some functions had to be defined in an awkward manner. We consider for instance the effect function:

```
definition effect :: "'a ts ⇒ 'a operator ⇒ 'a state ⇒ 'a state" where
"effect T op s ≡ (SOME s' :: 'a state. s' ∈ states T ∧ (∃ t ∈ op. (s = src t ∧ s' = dst t)))"
```

For cases where the `op` is not valid, `effect` will map the input to more than one output, namely to a class states satisfying the definition, and if `op` is not applicable in `s`, `effect(op, s)` is ill defined. Hence the `SOME` keyword. Of course, we only care about the cases where `op` is valid and applicable in `s`, then `effect` behaves like a function in the mathematical sense, but this requires proof, and although this is obvious, the proof is not easy. Another approach could have been to follow the functional programming paradigm and define a function taking a transition system as an input that returns an effect function with a restricted co-domain of valid operators; but this too has its issues.

In the next section now present a small highlight of the Isabelle/HOL proof.

5.1 A Note on the pull function

Corollaries 1 and 2 are one-line proofs, albeit there are intricate mathematics behind the scenes, and in Isabelle they cannot be ignored. Among others, the proofs use the fact that if op is in π , and if the length of π is n , then the chain of compositions over swp of length $n - 1$ evaluated at π and op has op at its first element. Intuitively, we can justify this as follows: if the index of op in π is I , with the indexing starting at 0, then the index of op in $swp(\pi, op)$ is $\max\{I - 1, 0\}$, so the index of op in the chain $swp(swp(\dots swp(\pi, op), \dots), op)$ is $\max\{I - n, 0\}$, and since $I < n$, this evaluates to 0. This argument is very convincing and may be claimed as trivial in a scientific paper or a textbook, but without some further scrutiny, it is just not rigorous enough for Isabelle to accept it.

We present an overview of our Isabelle proof. Before jumping into it, we provide the necessary definitions.

From now on we use square brackets to enclose sequences instead of \langle and \rangle to stick to syntax of Isabelle/HOL. We also call sequences lists, denote the first element of a list π by $hd(\pi)$, its length by $len(\pi)$, and re-use the definition of the \cdot function from Chapter 4.1.

The swapping function, called `swap_with_previous` in the proof (but shortened to `swp` in this text) is defined recursively as follows:

$$\text{swp}(\pi, op) = \begin{cases} \pi & \text{if } len(\pi) \leq 1 \vee hd(\pi) = op \\ [y, x] \cdot \mathbf{xs} & \text{if } \pi = [x, y] \cdot \mathbf{xs} \text{ for some } \mathbf{xs}, x, y : x \neq op \wedge y = op \\ [x] \cdot \text{swp}([y] \cdot \mathbf{xs}, op) & \text{if } \pi = [x, y] \cdot \mathbf{xs} \text{ for some } \mathbf{xs}, x, y : x \neq op \wedge y \neq op \end{cases}$$

For the chaining of compositions of `swp` we define the function `pull` as a map that takes a list of elements of some type `'a`, a natural number and an element of type `'a` as arguments and returns a list of elements of type `'a`, where

$$\text{pull}(\pi, n, op) = \begin{cases} \pi & \text{if } n = 0 \\ \text{pull}(\text{swp}(\pi, op), n - 1, op) & \text{if } n > 0 \end{cases}$$

`pull` is essentially a for-loop that executes $\pi = swp(\pi, op)$ on every iteration. Now it is clear what must be shown:

$$op \text{ in } \pi \implies hd(\text{pull}(\pi, len(\pi), op)) = op \quad (5.1)$$

The idea of using indices demonstrated with the informal argument has been tried and later abandoned since proving that the index function is weakly decreasing under compositions of `swp` chain is just as difficult as proving the claim above.

Instead, the claim is proven by induction on π : the base case works fine, however for the inductive step, because π appears twice inside the `pull` arguments, this leads to an awkward state of affairs, and it is difficult to make progress directly. The key idea is to first prove the lemma:

$$op \text{ in } \pi \implies hd(\text{pull}(\pi, n, op)) = op \implies hd(\text{pull}([x] \cdot \pi, n + 1, op)) = op \quad (5.2)$$

By natural induction on n , as now there is only one argument of `pull` for which induction is being applied to. 5.1 then follows from 5.2 without many complications.

The proof of this lemma is shown in Figure 5.3. In it, we can see that this lemma also depends on more elementary lemmas about the behavior of `pull` and `swp`, namely

- `pull_head_to_head_does_nothing` on line 513, that claims that if $hd(\pi) = op$, then $hd(\text{pull}(\pi, n, op)) = op$.
- `head_concat_swap` on line 522, that claims that if op is in π and $op \neq x$ and $op \neq hd(\pi)$ then $[x] \cdot \text{swp}(\pi, op) = \text{swp}([x] \cdot \pi, op)$.
- `set_swap_eq` on line 531 that claims that $\text{set}(\pi) = \text{set}(\text{swp}(\pi, op))$.

Among others. Some of these lemmas themselves are proven by using yet more elementary lemmas about the behavior of `swp` and `pull`.

```

590 lemma pull_suc_n: "op ∈ set xs ⇒ hd (pull xs n op) = op ⇒ hd (pull (x#xs) (Suc n) op) = op"
591 proof(induct n arbitrary: xs)
592   case 0 then show ?case by (metis empty_iff list.sel(1) neq_Nil_conv pull.simps(2)
593     pull_eq_repeat_swap repeat.simps(1) set_empty2 swap_with_previous.simps(3))
594 next
595   case (Suc n)
596   have 1: "op ∈ set xs ⇒ hd (pull xs (Suc n) op) = op ⇒ hd (pull (swap_with_previous xs op) n op) = op" by auto
597   have 2: "op ∈ set xs ⇒ hd (pull (x#xs) (n+2) op) = op ⇒ hd (pull (swap_with_previous (x#xs) op) (n+1) op) = op" by auto
598
599   from 1 and 2 and local.Suc and pull_head_to_head_does_nothing and swap_with_previous.simps(3) and
600     list.set_cases pull.simps(2) and list.sel(1) have 3: "op ∈ set xs ⇒ hd (pull (swap_with_previous xs op) n op) = op
601     ⇒ hd (pull (x#(swap_with_previous xs op)) (Suc n) op) = op ⇒ ?case" by smt
602
603   from pull_head_to_head_does_nothing have ONE: "op ∈ set xs ∧ x = op ⇒
604     hd (pull (swap_with_previous xs op) n op) = op ⇒ hd (pull (x#(swap_with_previous xs op)) (Suc n) op) = op"
605     by (metis list.sel(1))
606
607   have TWO: "op ∈ set xs ∧ x ≠ op ∧ hd xs = op ⇒ hd (pull (swap_with_previous xs op) n op) = op ⇒
608     hd (pull (x#(swap_with_previous xs op)) (Suc n) op) = op"
609     by (metis Suc.hyps Suc.prem(1) swap_head_with_prev)
610
611   have "op ∈ set xs ∧ x ≠ op ∧ hd xs ≠ op ⇒ x # (swap_with_previous xs op) = swap_with_previous (x#xs) op"
612     using head_concat_swap by fastforce
613
614   from this and 3 have 4: "op ∈ set xs ∧ x ≠ op ∧ hd xs ≠ op ⇒ hd (pull (swap_with_previous xs op) n op) = op
615     ⇒ hd (pull ((x#swap_with_previous xs op)) (Suc n) op) = op ⇒ ?case" by simp
616
617   from local.Suc have "op ∈ set (swap_with_previous xs op) ∧ x ≠ op ∧ hd (swap_with_previous xs op) ≠ op
618     ⇒ hd (pull (swap_with_previous xs op) n op) = op
619     ⇒ hd (pull (x#(swap_with_previous xs op)) (Suc n) op) = op" by simp
620
621   from this show ?case using "3" Suc.hyps Suc.prem(1) Suc.prem(2) set_swap_eq by force
622 qed

```

Figure 5.3: An Isabelle/HOL proof of 5.2.

6

Conclusion

We have managed to validate the correctness of the optimality preserving property of Strong Stubborn Set based pruning in the setting of transition systems. We now have a higher degree of confidence that this pruning procedure is safe to use.

During the validation of Theorem 2 in Isabelle/HOL, a common pitfall has been to rigorously define the theory of Strong Stubborn Sets and to prove facts about these, since in the classical planning literature, most of the definitions are given at a meta-mathematical level, which limits the high level of granularity that may be used to prove facts about them. This further emphasizes the interest in computer assisted validations of important theorems in classical planning.

The topic of Strong Stubborn Set-finding algorithms has not been discussed in this thesis. Future work includes the the validation of the correct behavior of such algorithms. Another way in which this work can be expanded on is by validating Theorem 1 in Isabelle/HOL directly, without deviating to transition systems, since avoiding this reformulation reduces the potential for errors.

Bibliography

- [1] Isabelle’s archive of formal proofs. <https://www.isa-afp.org/>.
- [2] Isabelle mirror. <https://github.com/isabelle-prover/mirror-isabelle>.
- [3] Yusra Alkhazraji, Martin Wehrle, Robert Mattmüller, and Malte Helmert. A stubborn set algorithm for optimal planning. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*, pages 891–892, 2012.
- [4] Neil Burch and Robert C Holte. Automatic move pruning revisited. In *SOCS*, 2012.
- [5] Patrice Godefroid. Partial-order methods for the verification of concurrent systems — an approach to the state-explosion problem. volume 1032, 1996.
- [6] 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.
- [7] Malte Helmert, Gabriele Röger, et al. How good is almost perfect?. In *AAAI*, volume 8, pages 944–949, 2008.
- [8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220, 2009.
- [9] Martin Wehrle and Malte Helmert. About partial order reduction in planning and computer aided verification. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

Declaration on Scientific Integrity

Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor

Travis Rivera Petit

Matriculation number — Matrikelnummer

2015-117-427

Title of work — Titel der Arbeit

A Formal Verification of Strong Stubborn Set Based Pruning

Type of work — Typ der Arbeit


Bachelor's 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, 09.05.2020



Signature — Unterschrift