FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Space & Congruence Compression of Proofs

Diplomarbeit

im Rahmen des Studiums

European Master in Computational Logic

eingereicht von

Andreas Fellner, BSc
Matrikelnummer 0825918

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ. Prof. Dr.phil. Alexander Leitsch
Mitwirkung: Bruno Woltzenlogel Paleo, Dr.

Wien, 14. August 2014

(Unterschrift Andreas Fellner, BSc)	(Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Space & Congruence Compression of Proofs

Master thesis

in

European Master in Computational Logic

by

Andreas Fellner, BSc
Registration Number 0825918

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ. Prof. Dr.phil. Alexander Leitsch
Assistance: Bruno Woltzenlogel Paleo, Dr.

Vienna, 14. August 2014 ______________________ ______________________
(Signature of Author) (Signature of Advisor)

Abstract

This work is about compression of formal proofs. Formal proofs are of great importance to modern computer scienece. They can be used to combine deductive systems. For example SAT- Solvers [8] are heavily used for all kinds of computations, because of their efficiency. A formal proof is a certificate of the correctness of the output of a SAT- Solver. Furthermore, from formal proofs information can be extracted about some underlying problem. For example, Interpolants [25] can be extracted from proofs, as done in [23].

Typically problems tackled by automated systems are huge. Therefore the produced proofs are huge. For example, [24] reports about a 13 GB proof of one case of the Erdős Discrepancy Conjecture. With such proof sizes, computer system reach their boundaries and that is why it is necessary to compress proofs. Our work presents two methods for proof compression.

The first method removes redundancies in the congruence part of SMT- proofs. Congruence reasoning deduces equations from a set of given equations, using the four axioms *reflexivity*, *symmetry*, *transitivity*, and *congruence*. We found that SMT- Solver often use an unnecessarily big set of input equations to deduce one particular equality. We want to find smaller sets of equations, that suffice to proof the same result and therefore replace subproofs with shorter ones. Furthermore, we will proof the NP - Completeness of the problem of finding the shortest explanation of one equation within a set of input equations.

The second method investigates the memory requirements of proofs. While processing a proof, not all parts of the proof have to be kept in memory at all times. Subproof can be loaded into memory when needed and can be removed from memory again when they are not. In which traversal order subproofs are visited is essential to the maximum memory consumption during proof processing. We want to construct traversal orders with low memory requirements using heuristics.

Kurzfassung

Diese Arbeit befasst sich mit der Komprimierung von formalen Beweisen. Formale Beweise sind von großer Bedeutung in der modernen Informatik. Sie können verwendet werden um deduktive Systeme miteinander zu kombinieren. Ein Beispiel sind SAT- Solver [8], welche ob ihrer Effektivität gerne für diverse Berechnungen verwendet werden. Ein formaler Beweis kann als Zertifikat für die Korrektheit des Ergebnisses eines SAT- Solvers dienen. Des Weiteren können aus ihnen Informationen, wie etwa Interpolants [25], extrahiert werden, welche zur Lösung eines Problems beitragen [23].

Formale Beweise sind typischerweise sehr groß, siehe etwa [24] für einen 13GB Beweis eines Falles der Erdős Discrepancy Conjecture. Bei solchen Beweisgrößen stoßen Computersysteme an ihre Grenzen und deswegen ist es erforderlich Beweise zu komprimieren. Unsere Arbeit präsentiert zwei Methoden zur Beweiskomprimierung.

Die erste Methode entfernt Redundanzen im Kongruenzteil von SMT-Beweisen. Kongruenzbeweise schließen von einer Menge an Gleichungen auf neue Gleichungen mit der Vorraussetzung der vier Axiome: *Reflexivität*, *Symmetrie*, *Transitivität* und *Kongruenz*. Beweise, die von SMT-Solvern erzeugt werden, schließen oft auf neue Gleichungen aus einer unnötig großen Menge. Wir wollen kleinere Mengen finden, die für den Beweis der selben Aussage ausreichen und somit redundante Beweise durch kürzere ersetzen. Außerdem werden wir die NP - Completeness des Problems der kürzesten Erklärung einer Gleichung beweisen.

Die zweite Methode untersucht die Speicherplatzanforderungen von Beweisen. Beim Bearbeiten von Beweisen muss nicht der gesamte Beweis zu jeder Zeit im Speicher gelagert werden. Teilbeweise werden erst in den Speicher geladen, wenn sie benötigt werden und werden wieder aus diesem entfernt, sobald sie nicht mehr benötigt werden. In welcher Ordnung die Teilbeweise geladen werden, ist essentiell für die maximale Speicherplatzanforderung. Wir wollen Ordnungen mit niedrigen Speicherplatzanforderungen mit Hilfe von Heuristiken konstruieren.

Contents

CHAPTER 1

Introduction

Proofs are the backbone of mathematics. They allow scientists to build theorems on top of another and thus discover new knowledge. Proofs not only serve as stepping stones, they can also provide insight on the nature of the underlying problem.

Both statements are true for formal proofs, i.e. proofs in a formal calculus, as well. Formal proofs allow systems to trust the output of other systems and therefore they can safely be built on top of another. For example SAT-Solvers are used extensively in modern deductive systems [8]. However, solvers may contain bugs. Therefore their output can not be trusted blindly. A formal proof can assure the correctness of the output. Formal proofs not only help in combining systems, they can also be used to obtain information about the underlying problem. For example interpolants, which have important applications in Verification and Synthesis of programs [25], can be extracted from formal proofs [23]. Since this work is concerned only with formal proofs, from hereon we mean formal proofs when speaking of proofs.

Typically problems that are tackled by automated systems are large. As a consequence proofs produced during the process are large. So large that even for proof processing algorithms with low complexity, it is highly desirable to reduce the hardness of the input, while maintaining the quality of its conclusion. Proof processing algorithms could be correctness checking, information extraction or proof manipulating techniques, like for example proof compression. We present methods to compress proofs, produced by SMT- or SAT- Solvers, w.r.t. two different measures. Due to the enormous size of proofs our methods were constructed with the goal of low complexity in runtime.

The first measure we want to reduce is *length*. The length of a proof is the number of inferences. For example the length of the resolution proof is the number of applications of the resolution rule. The compression method we present is applicable to proofs in the SMT theory of equality. The congruence reasoning part of SMT-proofs has often been found to be redundant. Congruence reasoning derives equations of terms that are implied by a given set of input equations, using the four axioms *reflexive*, *symmetric*, *transitive* and *compatible*. It can be redundant in the sense that subsets of the input may suffice to derive certain equalities. In chapter **??** we present resolution proofs extended by equality and a method to compress them using congruence closure. Furthermore we show that finding the shortest set of input equations explaining a given equality is NP-complete. This indicates that there is no efficient algorithm to compute the shortest explanation efficiently. Therefore we propose ideas and methods to obtain short explanations, while not blowing up in complexity. We present a new congruence closure algorithm, which runs in the best known asymptotic runtime. One benefit of our algorithm is the possibility to implement it using immutable data structures. Such data structures are a central concept in functional programming languages and it is often hard or impossible to translate an imperative description of an algorithm into a functional implementation.

The second measure we compress is *space*. Typically proofs can be represented as directed acyclic graphs. The space of a proof is the maximal number of nodes of that graph that have to be kept in memory at once while processing it. In chapter **??** we present a method to compress resolution proofs in their space measure. The problem of finding the lowest space measure of a proof can be reduced to finding the optional strategy in a pebbling game. For this game it was proven that constructing the best strategy is NP-complete. Just like for our length compression algorithm, we want an algorithm with a lower complexity. Therefore we propose a heuristic method and arguments why our heuristics are reasonable.

Both methods have been implemented into the proof compression software Skeptik and were evaluated on a big number of proofs, produced by the SMT-Solver VeriT. Most of the evaluation proofs are from the benchmarks of SMT-lib (http://smt-lib.org/). Additionally we evaluated on proofs that were used in [23] to synthesize boolean functions by extracting an interpolant from a single proof. The method in [23] has high complexity and is therefore heavily dependent on the size of the proof. Therefore compressing such proofs in reasonable time is a definite plus for their work and possible those of many others.

CHAPTER 2

Proof Compression

Proof Compression

This work presents two methods for proof compression. In this chapter we will explain what we understand as proof compression. To this end, we will define the notion of a proof, specify measures of such objects that can be compressed and define how to process them.

2.1 Propositional Resolution Calculus

In this chapter, we will define the propositional resolution calculus. Resolution is one of the most well known automated inference techniques and goes back to Robinson [**?**]. While it a pretty simple calculus with just one inference rule, proofs in that calculus tend to become big. This property and its popularity make it a good target for applying proof compression to it.

Propositional resolution can be seen as a simplification of first-order logic resolution to propositional logic. For basics about propositional logic, we refer the reader to [**?**]. For an extensive discussion of first-order logic resolution, we refer the reader to [**?**].

Definition 2.1.1 (Literal and Clause)**.** A *literal* is a propositional variable or the negation of a propositional variable. The *complement* of a literal ℓ is denoted $\overline{\ell}$ (i.e. for any propositional variable p, $\overline{p} = \neg p$ and $\overline{\neg p} = p$). The set of all literals is denoted by $\mathcal{L}$. A *clause* is a set of literals. $\perp$ denotes the *empty clause*.

Clauses represent formulas by interpreting it as the disjunction of its literals. Sets of clauses represent formulas by interpreting them as the conjunction of the interpreted clauses. The propositional resolution calculus operates on propositional formulas in conjunctive normal form, which are formulas that are represented by a set of clauses.

Definition 2.1.2 (Resolvent)**.** Let C_1 and C_2 be two different clauses and ℓ be a literal, such that $\ell \in C_1$ and $\overline{\ell} \in C_2$. The clause $C_1 \setminus \{\ell\} \cup C_2 \setminus \{\overline{\ell}\}$ is called the *resolvent* of C_1 and C_2 w.r.t. ℓ.

The condition of C_1 and C_2 being different technically is not necessary. However if it is possible to resolve a clause with itself, this means that the clause contains both the positive and negative version of a variable and is therefore tautological (i.e. trivially satisfiable). Since the resolution calculus is refutational, i.e. it seeks to show unsatisfiability, such clauses are of no use and therefore we forbid them. In case it is possible to produce a resolvent of two clauses w.r.t. two different literals, no matter which literal is chosen, the resulting resolvent will tautological. Therefore the choice of literal to resolve on is not of interest to us. In terms of proof calculi, axioms of the propositional resolution calculus are clauses and the single rule of the calculus is to derive a resolvent from previously derived clauses or axioms. This work studies the syntactic and semantic structure of derivations in this calculus, which are formally defined in the following.

Definition 2.1.3 (Resolution Derivation and Refutation)**.** Let $F = \{C_1, \ldots, C_n\}$ be a set of clauses. The notion of a *resolution derivation* for F is defined inductively.

- $\langle C_1, \ldots, C_n \rangle$ is a resolution derivation for F.
- If $\langle C_1, \ldots, C_m \rangle$ is a resolution derivation for F then $\langle C_1, \ldots, C_{m+1} \rangle$ is a resolution derivation for F if C_{m+1} is a resolvent of C_i and C_j with $1 \leq i, j \leq m$.

A *resolution refutation* is a resolution derivation containing the empty clause.

The correctness of the resolution calculus can be formulated as the statement, that a propositional logic formula, represented as a set of clauses, is equivalent to all resolution derivations of it. Since the empty clause is unsatisfiable and sets of clauses are interpreted as a conjunction, a resolution derivation is unsatisfiable. Therefore a resolution refutation of F is a witness to the validity of $\neg F$. For an unsatisfiable formula, there are many different resolution refutations. They differ w.r.t. the order in which new resolvents are produced. The aim of proof compression is to find short refutations among all possible ones. We prefer a different view on proofs, which is more suited for the purpose of proof manipulation. In the following definition, we present proofs as labeled graphs.

Definition 2.1.4 (Proof)**.** A *proof* φ is a rooted labeled directed acyclic graph $\langle V, E, v, \mathcal{L} \rangle$. The labeling function $\mathcal{L}$ maps nodes to clauses and edges to literals. The designated node $v \in V$ is the root of the graph, i.e. it is a node without children and every node of the graph is a recursive ancestor of the node. Furthermore, a proof as to fulfill one of the following properties:

1. $V = \{v\}, E = \emptyset$
2. There are proofs $\varphi_L = \langle V_L, E_L, v_L, \mathcal{L}_1 \rangle$ and $\varphi_R = \langle V_R, E_R, v_R, \mathcal{L}_2 \rangle$ such that $v \notin (V_L \cup V_R)$, $\mathcal{L}_1(x) = \mathcal{L}_2(x)$ for every $x \in (V_L \cap V_R)$, $\mathcal{L}(v)$ is the resolvent of $\mathcal{L}(v_L)$ and $\mathcal{L}(v_R)$ w.r.t. some literal ℓ, for $x \in V_L : \mathcal{L}(x) = \mathcal{L}_1(x)$ and for $x \in V_R : \mathcal{L}(x) = \mathcal{L}_2(x)$, $V = (V_L \cup V_R) \cup \{v\}$, $E = E_L \cup E_R \cup \{(v_L, v), (v_R, v)\}$.

The node v is called the *root* of φ and $\mathcal{L}(v)$ its *conclusion*. In case 2, φ_L and φ_R are *premises* of φ and φ is a *child* of φ_L and φ_R. A proof ψ is a subproof of a proof φ, if there is a path from φ to ψ in the transitive closure of the premise relation. A subproof ψ of φ which has no premises is an *axiom* of φ. V_φ and A_φ denote, respectively, the set of nodes and axioms of φ. P_v^φ denotes the premises and C_v^φ the children of the subproof with root v in a proof φ. When a proof is represented graphically, the root is drawn at the bottom and the axioms at the top. The *length* of a proof φ is the number of nodes in V_φ and is denoted by $l(\varphi)$.

Note that since the labeling of premises have to agree on common nodes and edges, the definition of the labeling $\mathcal{L}$ is unambiguous. Also note that in case 2 of Definition 2.1.4 V_L and V_R are not required to be disjunct. Therefore the underlying structure of a proof is really a directed acyclic graph (DAG) and not simply a tree. Modern SAT- and SMT-solvers, using techniques of conflict driven clause learning, produce proofs with a general DAG structure [8, 9]. The reuse of proof nodes plays a central role in proof compression [19].

Example 2.1.1. Consider the propositional logic formula $\Phi := (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1)$. In clause notation, this formula is written as $\langle \{x_1, x_2, \neg x_3\}, \{x_1, x_2\}, \{x_1, x_3\}, \{\neg x_1\} \rangle$. By resolving the clauses $\{x_1, x_2, \neg x_3\}$ and $\{x_1, x_2\}$, we obtain the clause $\{x_1, \neg x_3\}$, which we can resolve with $\{x_1, x_3\}$ to obtain $\{x_1\}$. Finally, we obtain the empty clause $\perp$ by resolving $\{x_1\}$ with $\{\neg x_1\}$. The resulting proof is shown graphically in Figure 2.1. Figure 2.2 shows a proof of the same formula, which is longer than the one proof we presented.

$$\begin{tikzcd}
\{x_1, x_2, \neg x_3\} & & \{x_1, x_2\} & & \\
& \{x_1, \neg x_3\} \arrow[ul] \arrow[ur] & & \{x_1, x_3\} & \\
& & \{x_1\} \arrow[ul] \arrow[ur] & & \{\neg x_1\} \\
& & & \perp \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 2.1: Proof of Φ's unsatisfiability

$$\begin{tikzcd}
\{x_1, x_2, \neg x_3\} & & \{\neg x_1\} & & \{x_1, \neg x_2\} & & \{x_1, \neg x_3\} \\
& \{\neg x_2, x_3\} \arrow[ul] \arrow[ur] & & \{\neg x_2\} \arrow[ul] \arrow[ur] & & \{\neg x_3\} \arrow[ulll] \arrow[ur] & \\
& & \{x_3\} \arrow[ul] \arrow[ur] & & & & \\
& & & \perp \arrow[ul] \arrow[uurr] & & &
\end{tikzcd}$$

Figure 2.2: Another Proof of Φ's unsatisfiability

In Section 2.1 the notion of a proof in our sense was defined. The aim of this work is to make proof processing easier by minimizing proofs in the two measures space and length. Proof processing could be checking its correctness, manipulating it, like we do in this work extensively, or extracting information, like interpolants and unsat cores, from it. The following definition makes the notion of proof processing formal.

Definition 2.1.5 (Proof Processing)**.** Let φ be a proof with nodes V and T be an arbitrary set. A function $f : V \times T \times T \to T$ is a *processing function* if there is a function $g_f : V \to T$ such that for every $v \in V$ with $P_v^\varphi = \emptyset$ (i.e. v represents an axiom), $g_f(v) = f(v, t_1, t_2)$ for all

$\{t_1, t_2\} \subseteq T$. Let $\mathcal{F}$ be the set of processing functions. The *apply function* $\mathrm{ap} : V \times \mathcal{F} \rightarrow T$ is defined recursively as follows.

$$\mathrm{ap}(v, f) = \begin{cases} f(v, \mathrm{ap}(pr_1, f), \mathrm{ap}(pr_2, f)) & \text{if } v \text{ has premises } pr_1 \text{ and } pr_2 \\ g_f(v) & \text{otherwise} \end{cases}$$

Processing a node v with some processing function f means computing the value $\mathrm{ap}(v, f)$. *Processing a proof* means to process its root node. □

Example 2.1.2. Checking the correctness of a proof (i.e. checking for the absence of faulty resolution steps) can be checked in terms of the following processing function with $T = \{\top, \bot\}$ and $\wedge$ being the usual boolean and-operation.

$$f(v, w_1, w_2) = \begin{cases} \top & \text{if } v \text{ has no premises} \\ w_1 \wedge w_2 & \text{if the conclusion of } v \text{ is a resolvent} \\ & \quad \text{of the conclusions of its premises} \\ \bot & \text{otherwise} \end{cases}$$

Processing a proof with this processing function yields $\top$ *iff* the proof is a correct resolution proof. □

CHAPTER 3

Congruence compression

In this chapter we present a method to compress proofs in length. The method manipulates SMT proofs of the theory of equality. To this end, in Section 3 we extend the resolution calculus presented in Section 2.1 to handle equality and its axioms. The proof compression method is based on the idea to replace long explanations for the equality of two terms by shorter ones. In Section 3 show that finding the shortest explanation is NP-complete. In Section 3 we present our explanation producing congruence closure algorithm, which is applied in the proof compression algorithm presented in Section 7. Closing this chapter, we give an outlook of possible future work.

Definitions

In this section we summarize notions that are used throughout the chapter. Most importantly, we define the notion of congruence closure, which is an essential part of our proof compression method.

Definition 3.0.6 (Terms and Subterms)**.** Let $\mathcal{F}$ be a finite set of function symbols and $arity : \mathcal{F} \to \mathbb{N}$. A tuple $\Sigma = \langle \mathcal{F}, arity \rangle$ is called a *signature*. A function symbol with arity zero is called a *constant*, one with arity one is called a *unary* function symbol and one with arity 2 is called *binary*. For a given signature Σ, the set of *terms* $\mathcal{T}^{\Sigma}$ is defined inductively.

$$\begin{aligned}
\mathcal{T}_0^{\Sigma} &= \{a \in \mathcal{F} \mid arity(a) = 0\} \\
\mathcal{T}_{i+1}^{\Sigma} &= \{g(t_1, \ldots, t_n) \mid arity(g) = n \text{ and } t_1, \ldots, t_n \in \mathcal{T}_i\} \\
\mathcal{T}^{\Sigma} &= \bigcup_{i \in \mathbb{N}} \mathcal{T}_i
\end{aligned}$$

Let $f(t_1, \ldots, t_n) \in \mathcal{T}^{\Sigma}$, then $t_1, \ldots, t_n$ are *direct subterms* of $f(t_1, \ldots, t_n)$. The *subterm* relation is the reflexive, transitive closure of the direct subterm relation.

We will omit the index Σ, if it is clear from context.

Definition 3.0.7 (Equation)**.** Let $\mathcal{T}^\Sigma$ be a set of terms. An *equation* of $\mathcal{T}^\Sigma$ is a tuple of terms, i.e. an element of $\mathcal{T}^\Sigma \times \mathcal{T}^\Sigma$.

For a set of equations E we denote by $\mathcal{T}_E$ the set of terms used in E, i.e. the set $\{t \mid t \text{ is subterm of some } u \text{ such that for some } v : (u, v) \in E \text{ or } (v, u) \in E\}$.

Definition 3.0.8 (Congruence Relation)**.** Let $\mathcal{T}$ be a set of terms. A relation $R \subseteq \mathcal{T} \times \mathcal{T}$ is a congruence relation, if has the following four properties:

- reflexive: for all $t \in \mathcal{T} : (t, t) \in R$
- symmetric: $(s, t) \in R$ implies $(t, s) \in R$
- transitive: $(r, s) \in R$ and $(s, t) \in R$ implies $(r, t) \in R$
- compatible: f is a n-ary function symbol and for all $i = 1, \ldots, n (t_i, s_i) \in R$ implies $f(t_1, \ldots, t_n), f(s_1, \ldots, s_n) \in R$

Clearly every congruence relation is also an equivalence relation (which is a reflexive, transitive and symmetric relation). Therefore every congruence relation partitions its underlying set of terms $\mathcal{T}$ into congruence classes, such that two terms (s, t) belong to the same class if and only if $(s, t) \in R$. The relations $\emptyset$ and $\mathcal{T} \times \mathcal{T}$ are trivial congruence relations. In this work we are interested in congruence relations induced by sets of equations. In other words, we compute the partitioning of the terms such that two terms in the same partition are proven to be equal by the input set of equations. To this end we define the notion of congruence closure of a set of equations.

Definition 3.0.9 (Congruence Closure)**.** Let E be a set of equations. The set $E^* \supseteq E$ is called the *congruence closure* of E, if E^* is a congruence relation on $\mathcal{T}_E$ and for every congruence relation C, such that $C \supset E$ follows $C \supseteq E^*$. It is easily seen that congruence relations are closed under intersection. Therefore E^* always exists.
We write $E \models s \approx t$ if $(s, t) \in E^*$ and say that E is an *explanation* for $s \approx t$. We call a pair (s, t) in a congruence closure an *equality* and we call an equality of compound terms $f(a, b), f(c, d)$ such that $E \models a \approx c$ and $E \models b \approx d$ a *deduced equality*.

Proposition 3.0.1 (Properties of the $\models$-relation)**.** *The $\models$-relation is monotone: $E_1 \subset E_2$ and $E_1 \models s \approx t$ implies $E_2 \models s \approx t$ and transitive: $E \models s \approx t$ and $E \cup \{(s, t)\} \models u \approx v$ implies $E \models u \approx v$.*

Proof. Monotonicity follows from the fact that congruence closure of E_1 is contained in the congruence closure of E_2. Since the congruence closure of E^* is E^* itself, it follows that $E \models u \approx v$ if and only if $E^* \models u \approx v$. Since $(s, t) \in E^*$, clearly it is the case that $E^* = (E \cup \{(s, t)\})^*$. Therefore $E \cup \{(s, t)\} \models u \approx v$ implies $(u, v) \in E^*$, i.e. $E \models u \approx v$ or in other words, the $\models$-relation is transitive.

□

The idea of using congruence closure in proof compression is to replace big explanations, w.r.t. cardinality, by smaller ones.

Resolution extended with equality

Equality is a well researched topic in computational logic. Among the most prominent approaches to deal with this special predicate are first-order resolution with paramodulation [**?**], its extension the superposition calculus [**?**] and term rewrite systems [**?**]. We present equality in a framework that is closer to propositional resolution. The main theme of this work is resolution proof compression. Therefore we express equality reasoning in terms of resolution proofs. We extend the resolution calculus, presented in Section 2.1, with the axioms of equality. First we extend the notions of atoms, literals and clauses

Definition 3.0.10 (Equality atom, literal and clause)**.** Let $\mathcal{T}$ be a set of terms and let V be a finite set of propositional variables. The set of *equality atoms* is defined as $V \cup \mathcal{T} \times \mathcal{T}$. An *equality literal* is an equality atom e or a negated equality atom $\neg e$. An *equality clause* is a set of equality literals. For an equality clause C, we call the sets of equations $pos(C) := \{(u, v) \mid u = v \in C\}$ the *positive part* and $neg(C) := \{(u, v) \mid u \neg v \in C\}$ the *negative part* of C.

A set of equations can be interpreted as a set of clauses, if every equation is interpreted as the singleton clause containing just the equation itself. In the context of equality atoms, we write equations $(s, t) \in \mathcal{T} \times \mathcal{T}$ as $s = t$ and $s \neq t$ for its negated version. As usual, a clause is interpreted as the disjunction of its literals and a set of clauses is interpreted as the conjunction its clauses.

From hereon, we restrict our attention to sets of terms that, on top of constants, have at most one function symbol f, which is binary. We justify this restriction in Section 3. The axioms defining congruence relations have to be reflected in our extended resolution proofs. We achieve this by defining axiom schemas, that can be instantiated with concrete terms.

Definition 3.0.11 (Axioms of Equality)**.** In the following axioms schemas, the variables $x_1, \dots, x_n$ are placeholders for terms. By simultaneously replacing all variables by terms of some set $\mathcal{T}$, one obtains an equality clause, which we call an *instance w.r.t.* $\mathcal{T}$ of the respective axiom of equality.

- reflexivity: $\{x = x\}$
- symmetry: $\{x_1 \neq x_2, x_2 = x_1\}$
- transitivity: $\{x_1 \neq x_2, x_2 \neq x_3, \dots, x_{n-1} \neq x_n, x_1 = x_n\}$
- compatability: $\{x_1 \neq x_3, x_2 \neq x_4, f(x_1, x_2) = f(x_3, x_4)\}$

Next we will define the resolution calculus extended by congruence axioms.

Definition 3.0.12 (Resolution with Equality)**.** Let ℓ be an equality literal and C_1, C_2 be equality clauses such that $\ell \in C_1$ and $\neg\ell \in C_2$. The clause $C_1 \setminus \{\ell\} \cup C_2 \setminus \{\neg\ell\}$ is the *resolvent* of C_1 and C_2 with *pivot* ℓ.
Let $F = \{C_1, \ldots, C_n\}$ be a set of equality clauses and let E be the biggest subset of F, such that every clause in E is an equation. The notion of a *congruence derivation* for F is defined inductively.

- $\langle C_1, \ldots, C_n\rangle$ is a congruence derivation for F.
- If $\langle C_1, \ldots, C_m\rangle$ is a congruence derivation for F then $\langle C_1, \ldots, C_{m+1}\rangle$ is a congruence derivation for F if C_{m+1} is an instance w.r.t. $\mathcal{T}_E$ of an axiom of equality or C_{m+1} is a resolvent of C_i and C_j with $1 \leq i, j \leq m$.

A *congruence refutation* is a congruence derivation containing the empty clause.
Let $D = \langle C_1, \ldots, C_m\rangle$ be a congruence derivation. The longest subsequence $\langle C_{i_1}, \ldots, C_{i_k}\rangle$ of D, such that $C_{i_1}, \ldots, C_{i_k}$ all are instances of axioms of equality, is called the equality reasoning part of D.

Just like resolution derivations, congruence derivations can be visualized as directed acyclic graphs.

Proposition 3.0.2 (Sound- & Completeness)**.** *Let E be a set of equations and $s, t \in \mathcal{T}_E$, then $E \models s \approx t$ if and only if there is a congruence refutation for $E \cup \{\{s \neq t\}\}$*

Proof. The existence of a congruence refutation in case $E \models s \approx t$ is proven in terms of a proof producing algorithm, presented in Section **??**. This algorithm produces a congruence derivation with last clause $\{u_1 \neq v_1, \ldots, u_n \neq v_n, s = t\}$ such that $\{(u_i, v_i) \mid i = 1, \ldots, n\} \subseteq E$. Clearly this proof can be extended to a congruence derivation for $E \cup \{s \neq t\}$.

Suppose there is a congruence refutation $\langle C_1, \ldots, C_n\rangle$ for $E \cup \{\{s \neq t\}\}$. Clearly we can assume that the equality reasoning part of this derivation is the whole sequence. Since every clause in $E \cup \{\{s \neq t\}\}$ is singleton, none of its literals is in the resolvent of such a clause with any other clause. Therefore we can assume that there is a $m < n$ such that $\{C_1, \ldots, C_m\}$ only contains of instances of equality axioms and recursive resolvents of such clauses. Furthermore, since the whole sequence is a refutation, we can assume that C_m is such that $neg(C_m) \subseteq E$ and $pos(C_m) = \{(s, t)\}$.

We show by induction on the clause structure, that for every clause $C \in \{C_1, \ldots, C_m\}$ that $pos(C) = \{(u, v)\}$ for some $(u, v) \in \mathcal{T}_E$ and that $neg(C) \models u \approx v$. Suppose that C is an instance of an equality axiom. Clearly, the positive part contains of some equation (u, v) and $neg(C) \models u \approx v$ follows directly form the definition of congruence closure, and in case of the transitive axiom also from the transitivity of equality. Let C be obtained by resolving the clauses D_1 and D_2, such that $pos(D_i) = \{(u_i, v_i)\}$ and $neg(D_i) \models u_i \approx v_i$ for $i \in \{1, 2\}$. Suppose D_1 and D_2 were resolved using the equality literal $u_1 = v_1$ (the only other possibility is $u_2 = v_2$ and the cases are completely symmetric). Then $pos(C) = \{(u_2, v_2\}$ and $neg(C) = neg(D_1) \cup (neg(D_2) \setminus (u_1, v_1))$. Using the monotonicity of the $\models$-relation (Proposition 3.0.1) and the fact that $neg(D_1) \subseteq neg(C)$ and $neg(D_2) \subseteq neg(C) \cup \{(u_1, v_1)\}$, it follows that $neg(C) \models u_1 \approx v_1$ and $neg(C) \cup \{u_1, v_1\} \models (u_2, v_2)$. Using the transitivity of the $\models$-relation (Proposition 3.0.1), it follows that $neg(C) \models u_2 \approx v_2$, which is the desired result.

$\square$

NP-completeness of Short Explanation Decision Problem

In Section 3 the notion of an explanation is defined and it was mentioned that we want to find short explanations in order to compress proofs. In this chapter we show that one might have to search a while to find the shortest one, by proving that the problem of deciding whether there is an explanation of a given size is NP-complete. Our proof of NP-completeness reduces the problem of deciding the satisfiability of a propositional logic formula in conjunctive normal form to the short explanation decision problem. For basics about satisfiability of propositional logic formulae and assignments, we refer the reader to [8]. We begin by formally defining the problem.

Definition 3.0.13 (Short explanation decision problem)**.** Let $E = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ be a set of equations, $k \in \mathbb{N}$ and (s, t) be a target equation. The *short path decision problem* is the question whether there exists a set E' such that $E' \subseteq E$, $E' \models s \approx t$ and $|E'| \leq k$.

First we define how to translate propositional formulae into sets of equations.

Definition 3.0.14 (Congruence Translation)**.** Let Φ be a propositional logic formula in conjunctive normal form with clauses $C_1, \ldots, C_n$ using variables $x_1, \ldots, x_m$. The congruence translation E_Φ of Φ is defined as the set of equations $\mathit{Assignment} \cup \mathit{Pos} \cup \mathit{Neg} \cup \mathit{Connect}$, where

$$\begin{aligned}
\mathit{Assignment} &= \{(\hat{x}_j, \top_j), (\hat{x}_j, \bot_j) \mid 1 \leq j \leq m\} \\
\mathit{Pos} &= \{(\hat{c}_i, t_i(\hat{x}_j)) \mid x_j \text{ appears positively in } C_i\} \\
\mathit{Neg} &= \{(\hat{c}_i, f_i(\hat{x}_j)) \mid x_j \text{ appears negatively in } C_i\} \\
\mathit{Connect} &= \{(t_i(\top_j), \hat{c}_{i+1}), (f_i(\bot_j), \hat{c}_{i+1}) \mid 1 \leq i \leq n+m, 1 \leq j \leq m\}
\end{aligned}$$

For presentation purposes we define the following sets for every $i = 1, \ldots, n$ and $j = 1, \ldots, m$

$$\begin{aligned}
T_{ij} &= \{(\hat{c}_i, t_i(\hat{x}_j)), (\hat{x}_j, \top_j), (t_i(\top_j), \hat{c}_{i+1})\} \\
F_{ij} &= \{(\hat{c}_i, f_i(\hat{x}_j)), (\hat{x}_j, \bot_j), (f_i(\bot_j), \hat{c}_{i+1})\}
\end{aligned}$$

The following examples shows the congruence translation of a propositional formula and a subset of the translation, corresponding to a satisfying assignment. We use the standard notion of satisfiability and present variable assignments as sets of those propositional variables being mapped to true.

Example 3.0.3. Let $\Phi := (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2)$. Figure 3.1 shows the graphical representation of the equations in $\mathit{Pos}, \mathit{Neg}$ and $\mathit{Connect}$ for the congruence translation E_Φ of Φ. Figure 3.2 shows the subset $\mathit{Assignment}$ of E_Φ.

Let $\mathcal{I} := \{x_1, x_3\}$. It is easy to see that $\mathcal{I} \models \Phi$. Figure 3.3 shows a graphical representation of $\mathcal{I}$. Note that the satisfiability of Φ does not depend on x_2. Therefore replacing $\mathit{Assignment}$

with $Assignment' := Assignment \setminus \{(\hat{x}_2, \top_2)\} \cup \{(\hat{x}_2, \bot_1)\}$ in E_Φ leads to another explanation of $(\hat{c}_1, \hat{c}_4)$ of equal size. In the proof of Lemma 3.0.4 we exclude such ambiguous sets by introducing additional topological clauses.

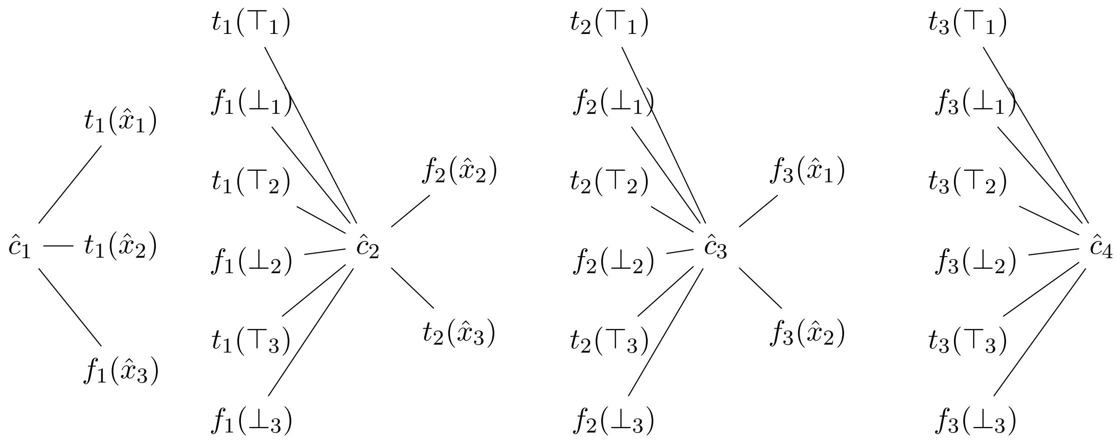

Figure 3.1: Pos, Neg and Connect for E_Φ

$$\begin{array}{c} \top_1 \;\text{——}\; \hat{x}_1 \;\text{——}\; \bot_1 \\[1em] \top_2 \;\text{——}\; \hat{x}_2 \;\text{——}\; \bot_2 \\[1em] \top_3 \;\text{——}\; \hat{x}_3 \;\text{——}\; \bot_3 \end{array}$$

Figure 3.2: Assignment for E_Φ

$$\hat{c}_1 \text{ — } t_1(\hat{x}_1) \text{ - - - } t_1(\top_1) \text{ — } \hat{c}_2 \text{ — } f_2(\hat{x}_2) \text{ - - - } f_2(\bot_2) \text{ — } \hat{c}_3 \text{ — } f_3(\hat{x}_2) \text{ - - - } f_3(\bot_2) \text{ — } \hat{c}_4$$

$$\begin{array}{ccccc} \top_1 & \text{————} & \hat{x}_1 & & \\[1em] & & \hat{x}_2 & \text{————} & \bot_2 \\[1em] \top_3 & \text{————} & \hat{x}_3 & & \end{array}$$

Figure 3.3: Explanation of $(\hat{c}_1, \hat{c}_4)$

Lemma 3.0.3 (Characterization of explanations). *Let Φ be a propositional logic formula in conjunctive normal form with n clauses and m variables. For every subset E of E_Φ, $E \models \hat{c}_1 \approx \hat{c}_{n+1}$ if and only if for every $i = 1, \ldots, n$ there is a $j = 1, \ldots, m$ such that $T_{ij} \subseteq E$ or $F_{ij} \subseteq E$.*

Proof. Suppose that for every $i = 1, \ldots, n$ there is a $j = 1, \ldots, m$ such that $T_{ij} \subseteq E$ or $F_{ij} \subseteq E$. Clearly $T_{ij} \models \hat{c}_i \approx t_i(\hat{x}_j)$ and $T_{ij} \models t_i(\top_j) \approx \hat{c}_{i+1}$. Since $(\hat{x}_j, \top_j) \in E$, the fact $E \models t_i(\hat{x}_j) \approx t_i(\top_j)$ follows by an application of the monotonicity axiom. Using the transitivity axiom it follows that $T_{ij} \models \hat{c}_i \approx \hat{c}_{i+1}$. Similarly it can be shown that $F_{ij} \models \hat{c}_i \approx \hat{c}_{i+1}$. Therefore it follows from the assumption that $E \models \hat{c}_i \approx \hat{c}_{i+1}$ for every $i = 1, \ldots, n$. Using the transitivity axiom it follows that $E \models \hat{c}_1 \approx \hat{c}_{n+1}$.
We will show the other direction of the equivalence by induction on n.

Induction Base $n = 1$: Suppose that $E \models \hat{c}_1 \approx \hat{c}_2$. Since $\hat{c}_1$ is a constant, the deduction axiom can not be applied to any equation with $\hat{c}_1$ on one side. Therefore in order to satisfy $E \models \hat{c}_1 \approx t$ with $t \neq \hat{c}_1$ there has to be an equation $(\hat{c}_1, t) \in E$ for some term t. Since $E \subseteq E_\Phi$, the only possible such equations are of the form $(\hat{c}_1, t_1(\hat{x}_j))$ and $(\hat{c}_1, f_1(\hat{x}_j))$ for some j. The only equations in E involving terms with the function symbols t_1 and f_1 are of the form $(\hat{c}_1, t_1(\hat{x}_j)), (t_1(\top_j), \hat{c}_2)$ and $(\hat{c}_1, f_1(\hat{x}_j)), (f_1(\perp_j), \hat{c}_2)$. Therefore in order to satisfy $E \models \hat{c}_1 \approx t$ such that t is neither the constant $\hat{c}_1$, nor some term $t_1(\hat{x}_j), f_1(\hat{x}_j)$, it is necessary that $E \models t_1(\hat{x}_j) \approx t_1(\top_j)$ and $(\hat{c}_1, t_1(\hat{x}_j)) \in E$ or $E \models f_1(\hat{x}_j) \approx f_1(\perp_j)$ and $(f_1(\perp_j), \hat{c}_2) \in E$ for some j. The conditions can only be satisfied with equations of E_Φ if $\{(\hat{c}_1, t_1(\hat{x}_j)), (\hat{x}_j, \top_j)\} \subseteq E$ or $\{(\hat{c}_1, f_1(\hat{x}_j), (\hat{x}_j, \perp_j)\} \subseteq E$ respectively. From a similar argumentation about the equations involving $\hat{c_2}$ and $t_1(\top_j)$ or $f_1(\perp_j)$ it follows that either $T_{1j} \subseteq E$ or $F_{1j} \subseteq E$ for some j.

Induction Hypothesis: For every subset E of E_Φ, $E \models \hat{c}_1 \approx \hat{c}_n$ if and only if for every $i = 1, \ldots, n-1$ there is a $j = 1, \ldots, m$ such that $T_{ij} \subseteq E$ or $F_{ij} \subseteq E$.

Induction Step: Suppose that $E \models \hat{c}_1 \approx \hat{c}_{n+1}$.
Similarly to the argumentation in the induction base, the only equations in E_Φ involving $\hat{c}_{n+1}$ are of the form $(t_n(\top_j), \hat{c}_{n+1})$ and $(f_n(\perp_j), \hat{c}_{n+1})$. The only possibility to enrich the congruence class of $\hat{c}_{n+1}$ with terms other than $\hat{c}_{n+1}$ and those of the form $t_n(\top_j)$ and $f_n(\perp_j)$, is that for some j, $(\hat{x}_j, \top_j) \in E$ or $(\hat{x}_j, \perp_j) \in E$ and subsequently also $(\hat{c}_n, t_n(\hat{x}_j)) \in E$ or $(\hat{c}_n, f_n(\hat{x}_j)) \in E$. Thus $T_{nj} \subseteq E$ or $F_{nj} \subseteq E$ and as a consequence $E \models \hat{c}_n \approx \hat{c}_{n+1}$. Using transitivity $E \models \hat{c}_1 \approx \hat{c}_{n+1}$ and $E \models \hat{c}_n \approx \hat{c}_{n+1}$ imply $E \models \hat{c}_1 \approx \hat{c}_n$ and from the induction hypothesis it follows that $T_{ij} \subseteq E$ or $F_{ij} \subseteq E$ for every $i = 1, \ldots, n-1$.

□

Lemma 3.0.4 (NP- hardness). *The short path decision problem is NP- hard.*

Proof. We reduce the problem of deciding the satisfiability of a propositional logic formula (better known as SAT) to the short path decision problem. SAT is a well known NP-complete problem [8]. Let Φ be a propositional formula in conjunctive normal form with variables $x_1, \ldots, x_m$ and clauses $C_1, \ldots, C_n$. Let $C_{n+1}, \ldots, C_{n+m}$ be the tautological clauses $\{x_1, \neg x_1\}, \ldots, \{x_m, \neg x_m\}$.

Clearly Φ is satisfiable if and only if $\Phi' = \{C_1, \ldots, C_{n+m}\}$ is satisfiable. We will show that Φ' is satisfiable if and only if there exists $E \subseteq E_{\Phi'}$ such that $E \models \hat{c}_1 \approx \hat{c}_{n+m+1}$ and $|E| \leq 2n + 3m$.

Suppose Φ' is satisfiable and let $\mathcal{I}$ be a satisfying assignment.
For every clause C_i there is a literal $\ell_i \in C_i$, such that $\mathcal{I} \models \ell_i$. For every $i = 1, \ldots, n+m$ we set $E_i := T_{ij}$ if $\ell_i = x_j$ and $E_i := F_{ij}$ if $\ell_i = \neg x_j$. From $\ell_i \in C_i$ follows $E_i \subseteq E_{\Phi'}$. Let $E = \bigcup_i^n E_i$ then from Lemma 3.0.3 and the transitivity of congruence relations follows $E \models c_1 \approx c_{n+m+1}$. What remains to show is that $|E| \leq 2n + 3m$. Since the sets Pos, Neg and $Connect$ in the definition of $E_{\Phi'}$ are pairwise disjoint, for $i \neq j$ $E_i \cap E_j \subseteq \{(\hat{x}_j, \top_j), (\hat{x}_j, \bot_j) \mid j = 1, \ldots, m\}$. Therefore E involves exactly $2(n+m)$ equations of Pos, Neg and $Connect$. By construction of the sets E_i and the clauses $C_{n+1}, \ldots, C_{n+m}$ there is no $j = 1, \ldots, m$ such that $(\hat{x}_j, \top_j) \in E$ and $(\hat{x}_j, \bot_j) \in E$. Therefore E involves m equations of set $Assignment$ in the definition of $E_{\Phi'}$. Overall we have $|E| = 2n + 3m$.

Suppose there exists $E \subseteq E_{\Phi'}$, $E \models \hat{c}_1 \approx \hat{c}_{n+m+1}$ and $|E| \leq 2n + 3m$.
We will show that $\mathcal{I} = \{\hat{x}_j \mid (\hat{x}_j, \top_j) \in E\}$ is a satisfying assignment for Φ'. Let $i = 1, \ldots, n+m$ be an arbitrary clause index. From $E \models \hat{c}_1 \approx \hat{c}_{n+m+1}$ and Lemma 3.0.3 follows $T_{ij} \subseteq E$ or $F_{ij} \subseteq E$ for some $j = 1, \ldots, m$.
Assume $T_{ij} \subseteq E$ for some $j = 1, \ldots, m$. $E \subseteq E_{\Phi'}$ implies that x_j appears positively in C_i. By definition of $\mathcal{I} \models x_j$. Therefore $\mathcal{I} \models C_i$.
If $T_{ij} \nsubseteq E$ for all $j = 1, \ldots, m$, then $F_{ij} \subseteq E \subseteq E_{\Phi'}$, which implies that x_j appears negatively in C_i, $x_j \notin \mathcal{I}$. Therefore $\mathcal{I} \models C_i$.
Since i was arbitrary $\mathcal{I} \models \Phi'$.

$\square$

Lemma 3.0.5 (In NP)**.** *The short explanation decision problem is in NP.*

Proof. Explanations are subsets of the input equations, therefore they are clearly polynomial in the problem size. The congruence of two terms, i.e. verifying that a subset is actually an explanation, can be decided in $O(nlog(n))$ using for example the congruence closure algorithm presented in Section 4.3.

$\square$

Lemma 3.0.4 and 3.0.5 establish the main result of this section.

Theorem 3.0.6 (NP - completeness)**.** *The short explanation decision problem is NP- complete.*

Explanation Producing Congruence Closure

In this section, we present a congruence closure algorithm that is able to produce explanations. The algorithm is a mix of the approaches of the algorithms presented in [18] and [27, 28]. The basic structure of the algorithm is inherited from [18], which itself inherits its structure from the algorithm of Nelson and Oppen [26]. The technique to store and deduce equalities of non constant terms is inspired from [27, 28]. Additionally the proof forest structure described below was proposed by [27, 28].

Preliminaries

Our congruence closure algorithm operates on curried terms. Curried terms use a single binary function symbol to represent general terms. More formally let $\mathcal{F}$ be a finite set of functions with a designated binary function symbol $f \in \mathcal{F}$ and let every other function symbol in $\mathcal{F}$ be a constant. A term w.r.t. a signature of this form is called a *curried term*.

It is possible to uniquely translate a general set of terms $\mathcal{T}^{\Sigma}$ with signature $\Sigma = \langle \mathcal{F}, arity \rangle$ into a set of curried terms $\mathcal{T'}^{\Sigma'}$. Σ' is obtained from Σ by setting $arity$ to zero for every function symbol in $\mathcal{F}$ and introducing the designated binary function symbol f to $\mathcal{F}$. The translation of a term $t \in \mathcal{T}^{\Sigma}$ is given in terms of the function $curry$.

$$curry(t) = \begin{cases} t & \text{if } t \text{ is a constant} \\ f(\ldots(f(f(g, curry(t_1)), curry(t_2)))\ldots, curry(t_n)) & \text{if } t = g(t_1, \ldots, t_n) \end{cases}$$

The idea of currying was introduced by M. Schönfinkel [33] in 1924 and independently by Haskell B. Curry [14] in 1958, who also lends his name to the concept. Currying is not restricted to terms. The general idea is to translate functions of type $A \times B \to C$ into functions of type $A \to B \to C$. There is a close relation between currying and lambda calculus [12]. Lambda calculus uses a single binary function λ. Its arguments can either be elements of some set or again lambda terms. For an introduction to lambda calculus, including currying in terms of lambda calculus and its relation to functional programming, see [3].

The benefit of working with curried terms is an easier and cleaner congruence closure algorithm, while maintaining best known runtime for congruence closure algorithms of $O(n \log(n))$. Cleaner algorithms are not only easier to implement, but should also improve the practical runtime.

Recently so called abstract congruence closure algorithms have been proposed and shown to be more efficient than traditional approaches [1]. The idea of abstract congruence closure is to introduce new constants for non constant terms. Doing so, all equations the algorithm has to take into account are of the form (c, d) and $(c, f(a, b))$, where a, b, c, d are constants. This replaces tedious preprocessing steps, for example transformation to a graph of outdegree 2 [16],that are necessary for other algorithms to achieve the optimal running time.

Our method is does not employ the idea of abstract congruence closure. We found that using currying is enough to obtain an algorithm with optimal running time and no tedious preprocessing steps. The reason why we did not go for abstract congruence closure is, that we do not want to have the overhead of introducing and eliminating fresh constants. In the context of proof compression, our congruence closure algorithm will be applied to relatively small instances very often. We could introduce the extra constants for the whole proof before processing, but would still have to remove them from explanations every time we produce a new subproof. It would be interesting to investigate, whether our intuition in that regard is right, or if it pays off to deal with extra constants.

Coming back to the explanation producing congruence closure algorithms that inspired our version, [27, 28] describes an version one using currying. [18] proposes a traditional algorithm without currying and extra constants. By choosing to work with curried terms, but without extra constants, our algorithm is a middle ground between them.

Congruence structure

We call the underlying data structure of our congruence closure algorithm a *congruence structure*. A congruence structure for set of terms $\mathcal{T}$ is a collection of the following data structures. The set $\mathcal{E} = \mathcal{T} \times \mathcal{T} \cup \{☺\}$ is the set of *extended equality*.

- Representative $r : \mathcal{T} \to \mathcal{T}$
- Congruence class $[.] : \mathcal{T} \to 2^{\mathcal{T}}$
- Left neighbors $N_l : \mathcal{T} \to 2^{\mathcal{T}}$ <- use N_l
- Right neighbors $N_r : \mathcal{T} \to 2^{\mathcal{T}}$
- Lookup table $l : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$
- Congruence graph g <– reference
- Queue $\mathcal{Q}$ of pairs of terms
- Current explanations $\mathcal{M} : \mathcal{T} \times \mathcal{T} \to \mathcal{E}$

The representative is one particular term of a class of congruent terms. It is used to identify whether two terms are in the same congruence class and the data structures used for detecting equalities derived from the compatability axiom (l, N_l and N_r) are kept updated only for representatives. The congruence class structure represents a set of pairwise congruent terms. It is used to keep track which representatives have to be updated when merging the classes of two terms. The structures left (resp. right) neighbor keeps track of the respective other terms in compound terms. The information is only used for representatives (i.e. terms in the target of $r(.)$). Furthermore right and left neighbors always only contain one term per congruence class (which is not necessarily the representative of that class). The lookup table is used to keep track of all compound terms in the congruence structure and to merge classes of compound terms, which arguments are congruent. For example if there terms $f(a,b), f(c,d)$ were inserted and the representatives are such that $r(a) = r(c), r(b) = r(d)$, then $N_r(r(a)) = \{d\}$, $N_l(r(d)) = \{a\}$ and $l(r(a), r(b)) = f(a,b)$. The elements in the respective sets serve as pointers to their representatives, therefore it does not matter whether for example $N_r(r(a)) = \{d\}$ or $N_r(r(a)) = \{b\}$. In Section 3 we explain how these structures are modified and used in detail. The congruence graph (explained in detail in Section 39) stores the derived equalities in a structured way, that allows to create explanations for a given pair of terms. Edges are added to the graph in a lazy way, meaning that they are buffered and only actually entered into the graph when demanded. The queue $\mathcal{Q}$ keeps track of the order in which edges should be added to the graph. The function $\mathcal{M}$ stores the existence of an explanation for a buffered edge. The idea will be explained in detail in Section 7. We call the unique congruence structure for $\mathcal{T} = \emptyset$ the *empty congruence structure*.

It is not by coincidence that many of the used data structures are described as functions. In fact our congruence closure algorithm can and is implemented in a functional way and the data structures can be implemented immutable.

Algorithm 3.1: addEquation

```
Input: equation (s, t)
addNode(s)
addNode(t)
merge(s, t, (s, t))
```

Congruence closure algorithm

In this section we present our congruence closure algorithm. We state and prove its properties. Most importantly we show that the algorithm is sound and complete and has the best known asymptotic running time $O(n \log(n))$. Computing the congruence closure of some set of equations E is done by adding all of them to an ever growing congruence structure, which initially is empty. Since this has to be done in some order, we will often assume that E is given as a sequence of equations rather than a set. The pseudocode of most methods do not include a return statement. In fact every method implicitly returns a (modified) congruence structure or simply modifies a global variable, which is the current congruence structure. Adding an equation to a congruence structure is done with the `addEquation` method. The method adds boths sides of the equation to the current set of terms using the `addNode` method and afterwards merges the classes of the two terms. The `addNode` method enlarges the set of terms and searches for compatible deduced equalities. The updates of the set of terms are not outlined explicitly, but are understood to happen implicitly. Throughout this chapter we denote this implicit set of terms by $\mathcal{T}$. The method `merge` initializes and guides the merging of congruence classes. The actual merging is done by the method `union` by modifying the data structures. The method does not only merge classes, but also searches for and returns equalities of compound terms that were caused by the merge and the compatible axiom. The classes of the terms of these extra equalities are merged, if they are not equal yet. The congruence classes are kept track of in a graph, maintaining important information for producing explanation and proofs. We call such a graph Congruence Graph and explain them in a more detailed fashion in Section 39. Edges, that reflect detected equalities, are not inserted into the graph right away, but stored in queue until the insertion is requested. The reason for adding edges in a lazy way is to produce shorter explanations and proofs and will be explained and exemplified in Section 7.

In the following pages, we will provide some invariants that are essential for proving the properties of the algorithm. The invariants hold when initializing the respective data structures and before and after every insertion of an equation via the `addEquation` method.

Invariant 3.0.7 (Class). *For every $s \in \mathcal{T}$ and every $t \in [r(s)]$, $r(t) = r(s)$.*

Proof. Clearly the invariant is true when intializing $[s]$ in line 3 of `addNode`.

The only other point in the code that changes $[s]$ is line 34 of union. Suppose the class of u is enlarged by the class of v in union and suppose the invariant holds before the union for those terms. Before the update of $[r(u)]$ the representative of every term in $[r(v)]$ is set to $r(u)$. Therefore the invariant remains valid after the update.

□

Algorithm 3.2: addNode

Input: term v

1 **if** *r is not defined for v* **then**
2 $\quad r(v) \leftarrow v$
3 $\quad [v] \leftarrow \{v\}$
4 $\quad lN(v) \leftarrow \emptyset$
5 $\quad rN(v) \leftarrow \emptyset$
6 $\quad$ **if** *v is of the form $f(a, b)$* **then**
7 $\quad\quad$ addNode(a)
8 $\quad\quad$ addNode(b)
9 $\quad\quad$ **if** *l is defined for $(r(a), r(b))$ and $l(r(a), r(b)) \neq f(a, b)$* **then**
10 $\quad\quad\quad$ merge($l(r(a), r(b)), f(a, b), ☺$)
11 $\quad\quad$ **else**
12 $\quad\quad\quad l(r(a), r(b)) \leftarrow f(a, b)$
13 $\quad\quad lN(r(b)) \leftarrow lN(r(b)) \cup \{a\}$
14 $\quad\quad rN(r(a)) \leftarrow rN(r(a)) \cup \{b\}$

Algorithm 3.3: lazy_insert

Input: term s
Input: term t
Input: extended equation eq

1 **if** *$\mathcal{M}$ is set for (s, t)* **then**
2 $\quad$ **if** $eq \neq ☺$ **then**
3 $\quad\quad \mathcal{M}(s, t) \leftarrow (s, t)$
4 **else**
5 $\quad \mathcal{Q} \leftarrow \mathcal{Q}.enqueue(s, t)$
6 $\quad \mathcal{M}(s, t) \leftarrow eq$

Algorithm 3.4: lazy_update

1 **while** *$\mathcal{Q}$ is not empty* **do**
2 $\quad (u, v) \leftarrow \mathcal{Q}.dequeue$
3 $\quad eq \leftarrow \mathcal{M}(u, v)$
4 $\quad g.insert(u, v, eq)$

Invariant 3.0.8 (Lookup)**.** *The lookup structure l is defined for a pair of terms (s, t) if and only if there is a term $f(a, b) \in \mathcal{T}$ such that $r(a) = r(s)$ and $r(b) = r(t)$.*

Proof. Suppose l is defined for some pair of terms (s, t). The value of $l(s, t)$ is set either in lines 33 or 19 of `union` or in line 39 of `addNode`. In the latter case, l is set to $f(a, b)$ for the tuple $(r(a), r(b))$ and therefore the invariant holds at this point. For changes to $r(a)$ or $r(b)$ in union the one implication of the invariant remains valid in case l is defined for the new representatives,

Algorithm 3.5: merge

Input: term s
Input: term t
Input: extended equation eq

```
if r(s) ≠ r(t) then
    c ← {(s, t)}
    eq ← (s, t)
    while c ≠ ∅ do
        Let (u, v) be some element in c
        c ← c \ {(u, v)} ∪ union(u, v)
        lazy_insert(u, v, eq)
        eq ← null
```

or l is set for an additional pair of terms in lines 19 or 33. In case l is set to $(new\_left, r(u))$ or $(r(u), new\_right)$ in union, there is an l-entry l_v for which the invariant held before the union. The changes in representatives of x are reflected by new_left and new_right, while the representative of v is changed to $r(u)$. The new entry for l therefore respects the implication of the invariant.

To show the other implication, let $f(a, b) \in \mathcal{T}$. The term $f(a, b)$ is entered via the `addEquation` method and subsequently via the `addNode` method. For compound terms lines and assert that l is defined for $(r(a), r(b))$. All changes to $r(a)$ or $r(b)$ must happen in `union` and they are reflected by matching updates to the l structure.

□

Invariant 3.0.9 (Neighbours)**.** *For every $s \in \mathcal{T}$, every $t_r \in rN(r(s))$ and $t_l \in lN(r(s))$, l is defined for $(r(s), r(t_r))$ and $(r(t_l), r(s))$.*

Proof. We show the result for the structure rN. The result about lN can be obtained analogously. Since rN is initialized with the empty set in line 5 of `addNode`, the invariant clearly holds initially. To show that the invariant always holds, it has to be shown that all modifications of r and rN do not change the invariant. The structure l is not modified after initialization. The structure r is modified in line 36 of union. The structure rN is modified in line 14 of addNode and line 38 of union.

Line 14 of `addNode` adds b to $rN(r(a))$ and the four lines before that addition show that l is defined for $(r(a), r(b))$.

Union modifies rN in such a way that it adds all right neighbors of some representative $r(v)$ to $rN(r(u))$. Lines 20 to 33 make sure that l is defined for all these right neighbors.

□

A consequence of this invariant is the fact that, that for every term $t \in \mathcal{T}$ of the form $f(a, b)$, l is defined for $(r(a), r(b))$.

Proposition 3.0.10 (Sound- & Completeness)**.** *Let $r(.)$ be the representative mapping obtained by adding equations $E = \langle(u_1, v_1), \ldots, (u_n, v_n)\rangle$ to the empty congruence structure. For every $s, t \in \mathcal{T}$: $E \models s \approx t$ if and only if $r(s) = r(t)$.*

Proof. **Completeness**

We show that from $E \models s \approx t$ follows $r(s) = r(t)$ by induction on n.

- **Induction Base** $n = 1$: $E \models s \approx t$ implies either $s = t$ or $\{u_1, v_1\} = \{s, t\}$. In the first case $r(s) = r(t)$ is trivial. In the second case, the claim follows from the fact that, when (u_1, v_1) is entered, union is called with arguments s and t. After this operation $r(s) = r(t)$.

- **Induction Hypothesis**: For every sequence of equations E_n with n elements and every $s, t \in \mathcal{T}_{E_n}$: $E_n \models s \approx t$ then $r(s) = r(t)$.

- **Induction Step**: Let $E = \langle(u_1, v_1), \ldots, (u_{n+1}, v_{n+1})\rangle$ and $E_n = \langle(u_1, v_1), \ldots, (u_n, v_n)\rangle$. There are two cases: $E_n \models s \approx t$ and $E_n \not\models s \approx t$. In the former case, the claim follows from the induction hypothesis, the invariant class and the fact that union always changes representatives for all elements of a class. We still have to show the claim in the latter case. We write $E \models_n u \approx v$ as an abbreviation for $E_n \not\models u \approx v$ and $E \models u \approx v$. We show the claim by induction on the structure of the terms s and t.

  - **Induction Base** s or t is a constant and therefore the transitivity reasoning was used to derive $E \models_n s \approx t$. In other words, there are l terms $t_1, \ldots, t_l$ such that $s = t_1$, $t = t_l$ and for all $i = 1, \ldots, l - 1 : E \models_n t_i \approx t_{i+1}$. We prove by yet another induction on l that $r(t_1) = r(t_l)$.

    * **Induction Base** $l = 2$. It has to be the case (up to swapping u_{n+1} with v_{n+1}), that $E_n \models s \approx u_{n+1}$ and $E_n \models t \approx v_{n+1}$, and the outmost induction hypothesis implies $r(s) = r(u_{n+1})$ and $r(t) = r(v_{n+1})$. Therefore it follows from Invariant Class, that after the call to union for (u_{n+1}, v_{n+1}) it is the case that $r(t_1) = r(t_2)$. Suppose that the claim holds for some $l \in \mathbb{N}$.

    * **Induction Step**: going from l to $l + 1$, the claim follows from a simple application of the transitivity axiom, since $t_1, \ldots, t_l$ and $t_2, \ldots, t_{l+1}$ are both sequences of length l.

  - **Induction Step**: suppose that $s = f(a, b)$ and $t = f(c, d)$. There are two cases such that $E \models_n s \approx t$ can be derived. Using a transitivity chain, the claim can be shown just like in the base case. Using the compatible axiom, it has to be the case that $E \models_n a \approx c$ and $E \models_n b \approx d$ (in fact one of those can also be the case without the n index). The terms a, b, c, d are of lower structure than s and t. Therefore it follows from the induction hypothesis that $r(a) = r(c)$ and $r(b) = r(d)$. The Invariants Neighbour and Lookup imply that either $r(s) = r(t)$ or (s, t) is added to d in line 15 or line line 29 of union. Subsequently union is called for s and t, after which $r(s) = r(t)$ holds.

Soundness

For $s = t$ the claim follows trivially. Therefore we show soundness in case $s \neq t$. We show that from $r(s) = r(t)$ follows $E \models s \approx t$ by induction on the number k of calls to union induced by adding all equations of E to the empty congruence structure for all s and t that are arguments of some call to union. The original claim then follows from invariant Class, since only union modifies the r structure and the fact that two terms are in the same class if and only if union was called for some elements in the respective classes.

- **Induction Base** $k = 1$: $r(s) = r(t)$ implies $\{u_1, v_1\} = \{s, t\}$ and $E \models s \approx t$ is trivial.
- **Induction Hypothesis**: For every $l < k$, if a set of equations F induces l calls to union, then from $r(s) = r(t)$ follows $F \models s \approx t$ for all terms s, t that are arguments of some call to union.
- **Induction Step**: Suppose $E = \langle (u_1, v_1), \dots, (u_n, v_n) \rangle$ induces k calls to union with arguments $(h_1, g_1), \dots, (h_k, g_k)$. The subsequence $E_n = \langle (u_1, v_1), \dots, (u_{n-1}, v_{n-1}) \rangle$ induced the first l calls to union for some $n-1 \leq l < k$. In other words, adding (u_n, v_n) to the congruence structure induces the calls to union with arguments $(h_{k-l}, g_{k-l}), \dots, (h_k, g_k)$. The first call to union with arguments (h_{k-l}, g_{k-l}) is either an original input equation, or a deduced equality from line 10 of `addNode`. In both cases $E \models h_{k-l} \approx g_{k-l}$, which is trivial in the former case and an application of the induction hypothesis in the latter case. Union induces additional union calls in such a way that the arguments of the additional call are on parent terms of the respective original arguments. Therefore, using induction on the structure of terms, the original induction hypothesis, Invariants Lookup and Neighbour and lines 6 to 33 of `union`, it can be shown that for all pairs (h_m, g_m) and all $m = k - l + 1, \dots, k$ it is the case that $E \models h_m \approx g_m$.

□

Proposition 3.0.11 (Runtime)**.** *Let E be a set of equations such that $|\mathcal{T}_E| = n$. Computing the congruence closure with our congruence closure algorithm takes worst-case time $O(n \log(n))$.*

Proof. There are three loops in the method `union`, which are nested within the loop of `merge`. These loops are clearly the dominating factor for runtime. Lines 2 and 4 of `union` swap the arguments s and t in such a way, that always the congruence class of v is smaller than the one of u. Let k be the size of the congruence class of v before the union. For every term in the congruence classes of v and u before the union, the size of their new congruence class after the union (set in line 34 is at least $2 * k$. Furthermore, only representatives for terms in the old congruence class of v are changed in line 36. This implies that for every term, whenever its representative is changed in line 36, its congruence class doubles in the same execution of `union`. The maximum size of a congruence class is n. Therefore the representative of a single term is changed in line 36 maximally $\log(n)$ times. There are n terms that can be changed, so line 36 of `union` is executed at most $n \log(n)$ times. Let $f(a, b)$ be the result of accessing l in line 7 In the same call of `union` line 36 changes the representative of b. Since this this happens only $\log(n)$ times and there are at most $m < n$ compound terms, line 7 is executed at most $n \log(n)$ times. The same holds for line 7 and all other lines in the respective loops.

□

Algorithm 3.6: union

```
Input: term s
Input: term t
Output: a set of deduced equations
if [r(s)] ≥ [r(t)] then
    (u, v) ← (s, t)
else
    (u, v) ← (t, s)
d ← ∅
for every x ∈ lN(r(v)) do
    l_v ← l(r(x), r(v))
    if r(x) = r(v) then
        new_left ← r(u)
    else
        new_left ← r(x)
    if l is defined for (new_left, r(u)) then
        l_u ← l(new_left, r(u))
        if r(l_u) ≠ r(l_v) then
            d ← d ∪ {(l_u, l_v)}
        else
            lN(r(v)) ← lN(r(v)) \ {x}
    else
        l(new_left, r(u)) ← l_v
for every x ∈ rN(r(v)) do
    l_v ← l(r(v), r(x))
    if r(x) = r(v) then
        new_right ← r(u)
    else
        new_right ← r(x)
    if l is defined for (r(u), new_right) then
        l_u ← l(r(u), new_right)
        if r(l_u) ≠ r(l_v) then
            d ← d ∪ {(l_u, l_v)}
        else
            rN(r(v)) ← rN(r(v)) \ {x}
    else
        l(r(u), new_right) ← l_v
[r(u)] ← [r(u)] ∪ [r(v)]
for every x ∈ [r(v)] do
    r(x) ← r(u)
lN(r(u)) ← lN(r(u)) ∪ lN(r(v))
rN(r(u)) ← rN(r(u)) ∪ rN(r(v))
return d
```

Congruence Graph

The main goal of this work is to replace redundant explanations with shorter ones. For this purpose the input equations and deduced equalities have to be stored in a data structure that supports the production of explanations. We support two different such data structures. Both structures store equalities in labeled graphs, which we call congruence graphs. A node in such a graph represents a term and an edge between two nodes denotes that the represented terms are congruent w.r.t. the set of input equations. A path in a congruence graph is a sequence of undirected, unweighted, labeled edges in the underlying graph. The set of labels for both types of graphs is the set of extended equalities $\mathcal{E}$. Depending on the type of congruence graph used, it is not guaranteed that after `lazy_insert` is called with arguments s and t, there is an edge between s and t. However it is guaranteed that the arguments are connected in the graph afterwards, i.e. there is a path between the nodes representing the terms.

Invariant 3.0.12 (Paths)**.** *For terms s, t such that $s \neq t$ and a congruence structure with representative function r holds $r(s) = r(t)$ if and only if there is a path in the congruence graph of the structure between s and t*

Proof. We show the claim by an induction on $|[r(s)]|$. The proof relies on the invariant Class, which shows the consistency between classes and representatives.

In the induction base $[r(s)] = \{s\}$, i.e. $r(s) = r(t)$ is false for every term $t \neq s$. We have to show that there is no edge (s, t) for $t \neq s$ in the congruence graph. Edges are only added to the congruence graph via the `lazy_insert` method which is only called in `merge`. Clearly `merge` does not call `union` for s and some term $t \neq s$, since otherwise $t \in [r(s)]$. Therefore `merge` also does not add an edge for s and some term $t \neq s$ to the congruence graph.

Let the induction hypothesis be, that for every term s such that $|[r(s)]| \leq n$, for every term $t \neq s$ it is the case that $r(s) = r(t)$ if and only if there is a path between s and t in the congruence graph.

Suppose $[r(s)]$ is an arbitrary class with cardinality $n + 1$. Then there are two terms $u, v \in [r(s)]$ such that `union` was called for u and v. Before the union $|[r(u)]|$ and $|[r(v)]|$ both were strictly smaller than $n+1$. In case they both belong to the same class before the union, the claim follows trivially by the induction hypothesis, since existing paths are not removed by adding new edges to the graph. Suppose $s \in [r(u)]$ and $t \in [r(v)]$, then by induction hypothesis there are paths p_1 between s and u and p_2 between t and v. Right after the union of u and v, an edge is inserted between them, so p_1 concatenated with (u, v) and p_2 is a path between s and t. In case one of the terms did not belong to one of the classes before the union, it does not belong to the merged class after the union. Also there was no path between the two terms before and since the only addition paths are between elements of $[r(u)]$ and $[r(v)]$, there is no path between the terms after the union.

□

Invariant 3.0.13 (Deduced Edges)**.** *For every edge in a congruence structure between vertices u, v with label ☺, there are $a, b, c, d \in \mathcal{T}$ such that $u = f(a, b)$, $v = f(c, d)$ and there are paths in the underlying graphs between a and c aswell as b and d.*

Proof. Edges with label ☺ are added, when `merge` is called from `addNode`, or `union` induces an additional merge. In both cases there are subterms with respective equal representatives. The claim follows by using the invariant Paths.

□

The method `explain` returns a path between its two arguments, if one exists. Depending on the actual type of graph used, this path can be unique or not. The method `inputEqs` for a path in the congruence graph returns the input equations that were used to derive the equality between the first and the last node of the path. For an input equation, this is simply the equation itself. For a deduced equality, this is the set of input equations that were used for deduction. Combining these two methods, the statement `inputEqs(explain(s,t,g),g)` returns an explanation for $E \models s \approx t$ if there is one.

Algorithm 3.7: inputEqs

Input: path p in g
Input: congruence graph g
Output: set of input equations used in p

Let p be $(u_1, l_1, v_1), \ldots, (u_n, l_n, v_n)$
$eqs \leftarrow \emptyset$ **for** $i \leftarrow 1ton$ **do**
  if $l_i =$ ☺ **then**
    $f(a, b) \leftarrow u_i$
    $f(c, d) \leftarrow v_i$
    $p1 \leftarrow \text{explain}(a, c, g)$
    $p2 \leftarrow \text{explain}(b, d, g)$
    $eqs \leftarrow eqs \cup inputEqs(p1, g) \cup inputEqs(p2, g)$
  else
    $eqs \leftarrow eqs \cup \{l_i\}$
return eqs

In the following, we describe the two types of congruence graphs we support. They differ in the type of graph they use and how explanations are produced.

Equation Graph

A equation graph stores input and deduced equalities in a labeled weighted undirected graph (V, E) with $V \subseteq \mathcal{T}$, $E \subseteq V \times \mathcal{E} \times V \times \mathbb{N}$. The weight for an edge is the number of input equalities used to derive the equality between its two nodes. This number is one for input equalities and the size of the explanation for deduced equalities. Edges are added to the graph, regardless whether the nodes are already connected in the graph. Therefore there is a choice which path the explain method returns. To produce short explanations, the shortest path w.r.t. the edge weights is returned.

Finding the shortest path between two nodes in a weighted graph is not trivial. The single source shortest path problem (SSSP) is a classical graph problem in computer science. The task is to find the shortest path in a graph between one designated node, the source, and all other nodes in the graph. To the best knowledge of the authors, there is no algorithm to find the shortest path between two nodes which has better asymptotic runtime than one to solve SSSP. There is a whole variety of algorithms that solve SSSP. Classical algorithms for SSSP are those of Dijkstra [15] and Bellman-Ford [4, 20]. The algorithms work on different kinds of graphs. Our setting is an undirected graph with positive integer weights. We chose to use Dijkstra's algorithm, even though the algorithm does not have optimal asymptotic runtime. It's worst-case runtime is $O(n \log(n))$ [13], if the priority queue is implemented as a Fibonacci Heap, which is the case in our implementation. [36] reports of an linear time algorithm for the undirected single source shortest path with positive integer weights problem. However, the algorithm has a big overhead and needs several precomputations. [11] is an extensive study of several shortest path algorithms which shows that Dijkstra's algorithm performs well in practice.

Dijkstra's algorithm finds shortest paths to an increasing set of nodes, until every node has been discovered. It does so by keeping track of the the shortest paths and the distances, being the combined weights of edges on the path, of nodes to the source. Initially, the only discovered node is the source itself and the distance to every other node is infinite. The algorithm discovers new nodes by selecting the lowest weight outgoing edge of all nodes that have been discovered so far and updates shortest paths and distances while doing so. It is a greedy algorithm in the sense that it always locally chooses lowest weight edges and never discards previously made decisions.

The algorithm has been slightly modified to take into account decisions that are edges for deduced equalities. These edges represent explanations, which are a sets of input equations. Previously included input equations do not increase the size of the global explanation when including them again. Therefore the modified Dijkstra algorithm adds an edge with weight 0 for every input equation in the explanation of a deduced equality edge. This is done to reduce the size of explanations. Since previous decisions are not discarded, it is not guaranteed that the modified algorithm returns the shortest path in the final graph, including the extra edges. Example 3.0.4 demonstrates that the modified shortest path algorithm does not always produce the shortest explanation, but can produce shorter explanations than the unmodified version in some situations. The shortest path algorithm's inability to return shortest explanations is not surprising, since it runs in $O(n \log(n))$ and in Section **??** it was shown that finding the shortest explanation is NP-complete.

Example 3.0.4. Consider the congruence graph shown in Figure 3.4, where solid edges are input equation and the dashed edge marks an application of the congruence axiom. The equality of $f(c_1, e)$ and $f(c_4, e)$ was deduced using the equations $(c_1, c_2), (c_2, c_3), (c_3, c_4)$, which is the shortest path in the graph between c_1 and c_4, obtained from a previous call to the shortest path algorithm.

Suppose we want to compute an explanation for $a \approx b$. Clearly the input equalities $(a, f(c_1, e))$, $(f(c_4, e), c_1)$ and the explanation for $f(c_1, e) \approx f(c_4, e)$ have to be included in the explanation. Additionally $c_1 \approx b$ has to be explained. For this equality the set $(c_1, d_1), (d_1, d_2), (d_2, b)$ is the shortest explanation in the original graph. This sub explanation adds three new equations to the explanation for $a \approx b$. Therefore when the shortest path algorithm iterates over the edge $(f(c_1, e), f(c_4, e))$, it can add add zero weight edges $(c_1, c_2), (c_2, c_3), (c_3, c_4)$ to the graph. By doing so the shortest explanation for $c_1 \approx b$ becomes $(c_1, c_2), (c_2, c_3), (c_3, c_4), (c_4, b)$, which only adds one extra equation to the global explanation.

This method is successful in finding the shortest explanation in this example if the search begins in the node a. Should the search begin in the node b, the edges including d_1, d_2 are added to the shortest path before the edge $(f(c_1, e), f(c_4, e))$ is touched. Therefore the undesired long explanation would be returned.

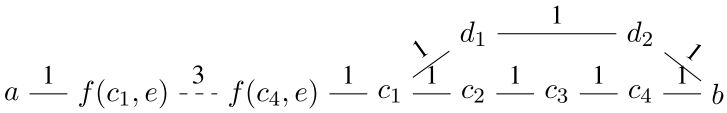

Figure 3.4: Short explanation example

Algorithm 3.8: insert (equation graph)

Input: term s
Input: term t
Input: extended equality $eq \in \mathcal{E}$

```
if eq ≠ ☺ then
    add edge (s, eq, t, 1) to g
else
    f(a, b) ← s
    f(c, d) ← t
    p1 ← shortest path between a and c in g
    p2 ← shortest path between b and d in g
    w ← #(p1.inputEqs ∪ p2.inputEqs)
    add edge (s, ☺, t, w)
```

Algorithm 3.9: explain

Input: term s
Input: term t
Input: equation graph g
Output: path in g

```
return shortest path between s and t in g
```

Proof Forest

A proof forest is a collection of proof trees. A proof tree is a labeled tree with nodes in $\mathcal{T}$ and edge labels in $\mathcal{E}$. For every congruence class in the current status of a congruence structure, there is one proof tree. Inserting an edge between nodes s and t of different proof trees is done by making one the child of the other. To maintain a tree structure, all edges between the new child and the root of its tree are reversed. To limit the number of edge revering steps, the smaller tree is always attached to the bigger one. This results in $O(n \log(n))$ edge reversing steps, where n is the number terms in the input equation set. This bound can be shown using the same argument as in the proof of Proposition 3.0.11. As stated earlier, we understand a path as a sequence of undirected edges. In case of a proof tree, a path between s and t of the same tree is the combined sequence of edges between the nodes and their nearest common ancestors. The structure, up to small changes, was proposed by [27, 28]. Its benefit is the quick access of explanations and good overall runtime. Its downside is its inflexibility when it comes to producing alternative explanations. In fact the explanation returned is always the first one to occur during edge insertion. The authors improve the structure for the special case of flattened terms, for which no term has nesting depth greater than one.

Algorithm 3.10: insert (proof forest)

Input: term s
Input: term t
Input: equation $eq \in \mathcal{E}$
if *s is not in g* **then**
  add tree with single node s
if *t is not in g* **then**
  add tree with single node t
$sSize \leftarrow$ size of tree of s
$tSize \leftarrow$ size of tree of t
if $sSize \leq tSize$ **then**
  $(u, v) \leftarrow (s, t)$
else
  $(u, v) \leftarrow (t, s)$
reverse all edges on the path between u and its root node
insert edge (v, eq, u)

Algorithm 3.11: explain

Input: term s
Input: term t
Input: proof forest g
if *s and t are in the same proof tree P* **then**
  Let nca be the nearest common ancestor of s and t in P
  $p1 \leftarrow$ path from s to nca
  $p2 \leftarrow$ path from nca to s
  return $p1 :: p2$
else
  return the empty path

Example 3.0.5. Consider again the set of equations presented in Figure 3.4 and Example 3.0.4 and suppose that the equations $(c_1, d_1), (d_1, d_2), (d_2, b)$ are inserted into the congruence structure before any other equation. After adding these three equations, the proof forest contains of a single proof tree and is displayed in Figure 3.5, where the labels are omitted. Suppose that next following equations are inserted: $(a, f(c_1, e)), (f(c_4, e), c_1), (c_1, c_2), (c_2, c_3)$. The resulting proof forest contains two proof trees and is shown in Figure 3.6. Finally the equation (c_3, c_4) is added and the equality $f(c_1, e) \approx f(c_4, e)$ is deduced. At this point, the explanation for $c_1 \approx c_4$ in the proof forest is the path $\langle c_1, c_2, c_3, c_4 \rangle$, which is the combined path from c_1 and c_4 to their nearest common ancestor, which is c_2. The resulting proof forest is shown in Figure 3.7, where the explanation for the edge $(f(c_1, e), f(c_4, e))$ is highlighted in a dotted rectangle. The explanation for $a \approx b$ in this graph is the path $\langle b, d_2, d_1, c_1, f(c_4, e), f(c_1, e), a \rangle$ and since the edge $(f(c_1, e), f(c_4, e))$ uses all other equations as explanation, the final explanation includes all eight equations. In example 3.0.4 we have shown that this is not necessary.

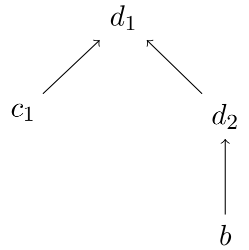

Figure 3.5: Proof Forest including first three equations

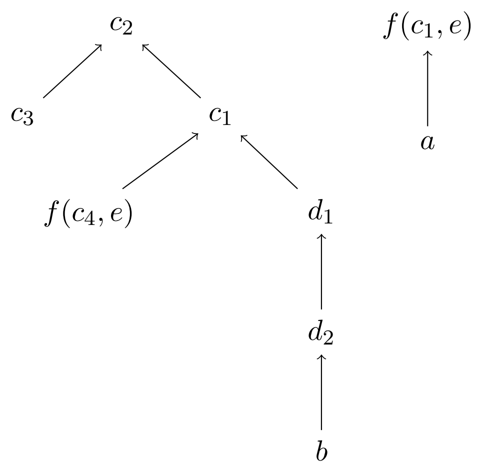

Figure 3.6: Proof Forest before deducing

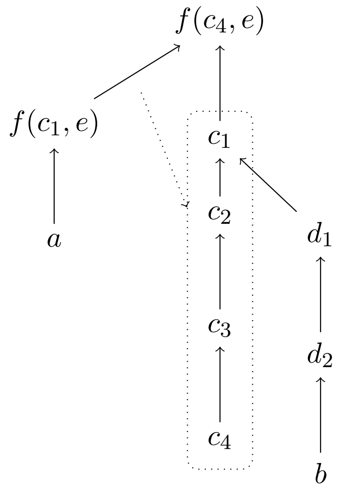

Figure 3.7: Final Proof Forest

Proof Production

In this section we describe how to produce resolution proofs from paths in a congruence graph. The method to carry out this operation is `prodcureProof`. The basic idea is to traverse the path, creating a transitivity chain of equalities between adjacent nodes, while keeping track of the deduced equalities in the chain. From invariant Deduced Edges follows that for the deduced equalities there have to be paths between the respective arguments of the compound terms. These paths are transformed into proof recursively and resolved with a suiting instance of the congruence axiom. Afterwards the subproof is resolved with the original transitivity chain. Since terms can never be equal to their subterms, the procedure will eventually terminate. The result of this procedure is a resolution proof with a root, such that the equations of the negative literals are an explanation of the target equality. In other words, let $s \approx t$ be the equality to be explained and suppose `produceProof` returns a proof with root ρ. Then for ρ it is the case that $F := \{(u, v) \mid u \neq v$ is a literal in $\rho\} \models s \approx t$ and F is a subset of the input equations.

Example 3.0.6. Consider again the congruence graph shown in Figure 3.4 and suppose we want a proof for $a \approx b$. Suppose we found the path $p_1 := \langle a, f(c_1, e), f(c_4, e), c_1, c_2, c_3, c_4, b\rangle$ as an explanation and that the explanation for $f(c_1, e) \approx f(c_4, e)$ is the path $\langle c_1, c_2, c_3, c_4\rangle$. We transform p_1 and p_2 into instances of the transitivity axiom C_1 and C_2 respectively. The clause C_2 is resolved with the instance of the congruence axiom C_3, which is then resolved with the instance of the reflexive axiom C_4 resulting in clause C_5. Finally, C_1 is resolved with C_5 to obtain the final clause C_6. The proof is shown in Figure 3.8.

As mentioned in Section 3, edges are inserted into a congruence graph in a lazy way by the congruence closure algorithm. The reason is that `produceProof` searches for explanations for edges with label ☺. Should the equality of question be an input equation that is added later to the congruence structure than it was deduced, then we would like to overwrite this label with

Algorithm 3.12: produceProof

Input: term s
Input: term t
Output: Resolution proof for $E \models s \approx t$ or $\emptyset$

$p \leftarrow explain(s, t, g)$
$d \leftarrow \emptyset$
$e \leftarrow \emptyset$
while p *is not empty* **do**
    $(u, l, v) \leftarrow$ first edge of p
    $p \leftarrow p \setminus (u, l, v)$
    $e \leftarrow e \cup \{u \neq v\}$
    if $l = ☺$ **then**
        $f(a, b) \leftarrow u$
        $f(c, d) \leftarrow v$
        $p_1 \leftarrow produceProof(a, c, g)$
        $p_2 \leftarrow produceProof(b, d, g)$
        $con \leftarrow \{a \neq c, b \neq d, f(a, b) = f(c, d)\}$
        $res \leftarrow$ resolve con with non $\emptyset$ roots of p_1 and p_2
        $d \leftarrow d \cup res$
if $\#e > 1$ **then**
    $proof \leftarrow e \cup \{s = t\}$
    while d *is not empty* **do**
        $int \leftarrow$ some element in d
        $d \leftarrow d \setminus \{int\}$
        $proof \leftarrow$ resolve $proof$ with int
    return $proof$
else if $d = \{ded\}$ **then**
    return ded
else
    if $e = \{(u, l, u)\}$ **then**
        return $\{u = u\}$
    else
        return $\emptyset$

the input equation. The impact of lazy insertion gets bigger, if an implementation searches for explanations already when an edge is added to the graph. Example **??** shows how this technique can help producing shorter proofs.

Example 3.0.7. Suppose we want to add the following sequence of equations into an empty congruence structure: $\langle (a, b), (f(a, a), d), (f(b, b), e), (f(a, a), f(b, b)) \rangle$. After adding the first three equations, the congruence closure algorithm detects the deduced equality $f(a, a) \approx f(b, b)$. The explanation for this equality is $\{(a, b)\}$, if we were to insert the edge $(f(a, a), f(b, b))$ into

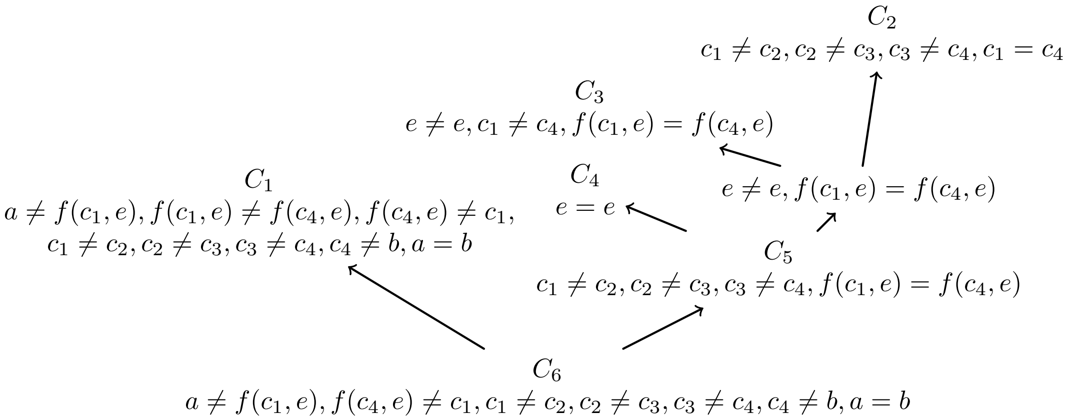

Figure 3.8: Example proof

the graph immediately, it would have weight 1 and label ☺. Depending on the congruence graph used, when adding the fourth equation $(f(a, a), f(b, b))$ to the congruence structure, either the edge $(f(a, a), f(b, b))$ is not added at all to the graph or is added with weight 1. In the latter case, both edges have weight 1 and equal chance to be selected by the shortest path algorithm. However, choosing the edge with label ☺ is undesirable, since it two extra resolution nodes (corresponding to the compatible axiom and an intermediate node).

Congruence Compressor

In Section **??** processing of a proof was defined. The most important application of proof processing for this work is proof compression. We want to make use of the short explanations found by the congruence closure algorithm described above. To this end we replace subproofs with conclusions that contain unnecessary long explanations with new proofs that have shorter conclusions. Shorter conclusions lead to less resolution steps further down the proof and possibly big chunks of the proof can simply be discarded. There is however a tradeoff in overall proof length when introducing new subproofs. The subproof corresponding to a short explanation can be longer in proof length, i.e. involve more resolution nodes, than one with a longer explanation. Example 3.0.8 displays this issue. Additionally it can be the case that by introducing a new subproof, we only partially remove the old subproof. Some nodes of the old subproof might still be used in other parts of the proof. Therefore the replacement of a subproof by another, smaller one does not necessarily lead to a smaller proof. Nevertheless, the meta heuristic favoring smaller conclusions should still dominate such effects, especially on large proofs. The results in Section **??** confirm this intuition.

Example 3.0.8. Consider the set of equations $E = \{(f(f(a, b), f(a, a)), a), (a, b), (b, f(f(b, a), f(b, b)))\}$ and the target equality $f(f(a, b), f(a, a)) \approx f(f(b, a), f(b, b))$. For presentation purposes, throughout this example we will abbreviate the term $f(f(a, b), f(a, a))$ with t_a and $f(f(b, a), f(b, b))$ with t_b. Using equations in E, one can prove the the target equality in two ways. Either one uses the instance of the transitivity axiom $\{t_a \neq a, a \neq b, b \neq t_b, t_a = t_b\}$ or a repeated applications of instances of the congruence axiom, e.g. $\{a \neq b, f(a,a) = f(b,b)\}$. The corresponding explanations are E and $\{(a,b)\}$.

The two resulting proofs are shown in Figure 3.9. The proof with the longer explanation E is only one proof node, whereas the proof with the singleton explanation has proof length 5.

Figure 3.9: Short explanation, long proof

The Congruence Compressor compresses processes a proof replacing subproofs as described above. It is defined upon the processing function $f : V \times V \times V \rightarrow V$ specified in pseudocode in Algorithm 13. The function $g_f : V \rightarrow V$ for axioms is simply the identity (i.e. axioms are not modified). The idea of the processing function is simple. Axioms are not changed by the function. For all other nodes the `fixNode` is called method, to maintain a correct proof. Then in line 2 it is decided whether the explanation finding congruence closure algorithm should be used to find a replacement for the current node. One trivial criteria could be true for every node. Testing every node will result in a slow algorithm, but the best possible compression. Some nodes do not need to be checked, since they contain optimal explanations by definition or there is no hope of finding an explanation at all. The following definition classifies nodes to define a more sophisticated decision criteria.

Definition 3.0.15 (Types of nodes)**.** An axiom is a *theory lemma* if it is an instance of one of the congruence axioms. Otherwise it is called *input derived*. The classification of internal nodes is defined recursively. An internal node is input derived, if one of its premises is input derived. Otherwise it is a theory lemma. We call a node a *low theory lemma* if it is a theory lemma and has a child that is input derived.

We suspect that most redundancies in proofs are to be found in low theory lemmas, since they reflect the explanations found by the proof producing solver. Therefore an alternative criteria is to only find replacements for low theory lemmas. The question whether a node is a low theory lemma is not trivial to answer while traversing the proof in a top to bottom fashion. Therefore a preliminary traversal is necessary to determine the classification of nodes. Experiments have shown that using this criteria speeds up the algorithm a lot, while losing only very little compression.

Add all equations of the antecedent to an empty congruence structure and check whether these equations induce a proof for one of the equations in the succedent that has a shorter conclusion than the original subproof. If there is such a proof, we replace the old subproof by the new one. Futher criteria for deciding whether to replace or not could be size of the subproof or a global metric that tries to predict the global compression achieved by replacement.

The compressor (Algorithm 13) uses the method `fixNode` to maintain a correct proof. The method modifies nodes with premises that have earlier been replaced by the compressor. Nodes with unchanged premises are not changed. Let n be a proof node that was derived using pivot ℓ in the original proof and which updated premises are pr_1 and pr_2 . Depending on the presence of ℓ in pr_1 and pr_2, n is either replaced by the resolvent of pr_1 and pr_2 or by one of the updated premises. It assumed that the values pr_1, pr_2 and ℓ are stored together with the node

Algorithm 3.13: compress

Global: set of input equations E
Input: resolution node n
Input: pr : tuple of resolution nodes (p_1, p_2)
Output: resolution node

$m \leftarrow fixNode(n, (p_1, p_2))$
if m *fulfills criteria* **then**
  $lE \leftarrow \{(a, b) \mid (a \neq b) \in m\}$
  $rE \leftarrow \{(a, b) \mid (a = b) \in m\}$
  $con \leftarrow$ empty congruence structure
  for (a, b) *in* lE **do**
    $con \leftarrow con.addEquality(a, b)$
  for (a, b) *in* rE **do**
    $con \leftarrow con.addNode(a).addNode(b)$
    $proof \leftarrow con.prodProof(s, t)$
    if $proof \neq \emptyset$ *and* $|proof.conclusion| < |m.conclusion|$ **then**
      $m \leftarrow proof$
return m

and can be accessed in constant time. In case both updated premises do not contain the original pivot element, replacing the node by either one of them maintains a correct proof. Since we are interested in short proofs, we return the one with the shorter clause. This method of maintaining a correct proof was proposed in [2] in the context of similar proof compression algorithms.

Algorithm 3.14: fixNode

Input: resolution node n
Input: pr : tuple of resolution nodes (p_1, p_2)
Output: resolution node

if $(n.premise_1 = p_1$ *and* $n.premise_2 = p_2)$ **then**
  return n
else
  if $n.pivot \in p_1$ *and* $n.pivot \in p_2$ **then**
    return $resolve(p_1, p_2)$
  else if $n.pivot \in p_1$ **then**
    return p_2
  else if $n.pivot \in p_2$ **then**
    return p_1
  else
    return node with smaller clause

Future Work

[1] compares the running times of several congruence closure algorithms. It would be interesting to do a similar comparison including the congruence closure algorithm presented in Section 3. A comparison to the classic congruence closure algorithms of Nelson and Oppen [26], Downey, Sethi and Tarjan [16] and Shostak [35] and their abstract counterparts, as described in [1], would show whether our method can compete in terms of computation speed. Comparing our method with the explanation producing algorithms presented in [18] and [27,28] could be done not only in terms of speed, but also in terms of explanation size.

In Section 3 it was shown that the problem of finding the shortest explanation is NP-complete. Therefore methods and heuristics to find short explanations could be investigated. The idea of using shortest path algorithms for explanation finding is a step in that direction. In **??** we describe a modification of Dijkstra's algorithm [**?**] to make it sensitive to previously used equations. Further modifications, possibly using heuristics, could lead to a short explanation algorithm. Furthermore translating the problem into a SAT instance could result in an algorithm to derive shortest explanations in acceptable time.

The congruence closure algorithm could be implemented into a SMT solver. Such solvers usually have high requirements regarding computation time. It would be interesting to see, whether the method presented in this work can match these requirements.

[28] extends the congruence closure algorithm to integer offsets; this could be incorporated

CHAPTER 4

Space compression

4.1 Pebbling Game

Pebbling games are played on directed acyclic graphs and pebbles are placed on nodes following the rules of the game. The goal is to put a pebble on some target node. Pebbling games were introduced in the 1970's to model programming language expressiveness [31, 38] and compiler construction [34]. More recently, pebbling games have been used to investigate various questions in parallel complexity [10] and proof complexity [6,17,29]. They are used to obtain bounds for space and time requirements and trade-offs between the two measures [5,37]. Space requirements are modeled with the number of pebbles used. Time requirements are reflected by the number of rounds played. From hereon *to pebble* means to mark a node with a pebble and *to unpebble* means to remove the mark off a node.

Definition 4.1.1 (Bounded Pebbling Game)**.** The *Bounded Pebbling Game* is played by one player on a DAG $G = (V, E)$ with one distinguished node $s \in V$. The goal of the game is to pebble s, respecting the following rules:

1. A node v is pebbleable *iff* all predecessors of v in G are pebbled and v is currently not pebbled.

2. Pebbled nodes can be unpebbled at any time.

3. Once a node has been unpebbled, it may not be pebbled in a later round.

The game is played in rounds. Every round the player chooses a node $v \in V$, such that v is pebbled or pebbleable. The move of the player in this round is $p(v)$, if v is pebbleable and $u(v)$ if v is pebbled, where $p(.)$ and $u(.)$ correspond to pebbling and unpebbling a node respectively. □

Not that due to rule 1 the move in each round is uniquely defined by the chosen node v. The distinction of the two kinds of moves is just made for presentation purposes. Also note that as a consequence of rule 1, pebbles can be put on nodes without predecessors at any time. Playing the game on a proof φ means to play the game on the underlying DAG with the distinguished node being the root of φ.

In this work we investigate space requirements when time requirements are fixed. Fixing time is a design choice, see Section **??**, and it corresponds to rule 3. Including this rules sets a bound $O(|V|)$ for the number of rounds.

Definition 4.1.2 (Strategy)**.** A *pebbling strategy* σ for the Bounded Pebbling Game, played on a DAG $G = (V, E)$ and distinguished node s, is a sequence of moves $(\sigma_1, \ldots, \sigma_n)$ of the player such that $\sigma_n = p(s)$.

Rules 2 and 3 The following definition allows to measure how many pebbles are required to play the Bounded Pebbling Game on a given graph.

Definition 4.1.3 (Pebbling number)**.** The *pebbling number of a pebbling strategy* $(\sigma_1, \ldots, \sigma_n)$ is $max_{i\in\{1\ldots n\}}|\{v \in V \mid v \text{ is pebbled in round } i\}|$. The *pebbling number of a DAG G and node* s is the minimum pebbling number of all pebbling strategies for G and s.

Note that Definitions 4.1.1 and 4.1.2 leave the player freedom when to do unpebbling moves. With the aim of finding strategies with low pebbling numbers, for every unpebbling move there is a canonical round make them, as will be shown in Section **??**.

The Bounded Pebbling Game from definition 4.1.1 differs from the Black Pebbling Game discussed in [22, 32] in two aspects. Firstly, the Black Pebbling Game does not include rule 3. Excluding this rule allows for pebbling strategies with lower pebbling numbers ([34] has an example on page 1), at the expense of an exponential upper bound on the number of rounds [37]. Secondly, when pebbling a node in the Black Pebbling Game, one of its predecessors' pebbles can be used instead of a fresh pebble (i.e. a pebble can be moved). The trade-off between moving pebbles and using fresh ones is discussed in [37]. Deciding whether the pebbling number of a graph G and node s is smaller than k is PSPACE-complete in the absence of rule 3 [21] and NP-complete when rule 3 is included [34].

4.2 Pebbling as a Satisfiability Problem

To find the pebble number of a proof, the question whether the proof can be pebbled using no more than k pebbles can be encoded as a propositional satisfiability problem. In this section let φ be a proof with nodes $v_1, \ldots, v_n$ and let v_n be its root node. Due to rule 3 of the Bounded Pebbling Game, the number of moves that pebble nodes is exactly n and due to theorem **??** determining the order of these moves is enough to define a strategy. For every $x \in \{1, \ldots, k\}$, every $j \in \{1, \ldots, n\}$ and every $t \in \{0, \ldots, n\}$ there is a propositional variable $p_{x,j,t}$. The variable $p_{x,j,t}$ being mapped to $\top$ by a valuation is interpreted as the fact that in the t'th round of the game node v_j is marked with pebble x. Round 0 is interpreted as the initial setting of the game before any move has been done.

Definition 4.2.1 (Pebbling SAT encoding)**.** The propositional formula obtained by conjuncting the following four constraints expresses the existence of a pebbling strategy for φ with pebbling number smaller or equal k.

1. The root is pebbled in the last round

$$\Psi_1 = \bigvee_{x=1}^{k} p_{x,n,n}$$

2. No node is pebbled initially

$$\Psi_2 = \bigwedge_{x=1}^{k} \bigwedge_{j=1}^{n} (\neg p_{x,j,0})$$

3. A pebble can only be on one node in one round

$$\Psi_3 = \bigwedge_{x=1}^{k} \bigwedge_{j=1}^{n} \bigwedge_{t=1}^{n} \left(p_{x,j,t} \rightarrow \bigwedge_{i=1, i \neq j}^{n} \neg p_{x,i,t} \right)$$

4. For pebbling a node, its premises have to be pebbled the round before and only one node is being pebbled each round.

$$\Psi_4 = \bigwedge_{x=1}^{k} \bigwedge_{j=1}^{n} \bigwedge_{t=1}^{n} \Bigg(\left(\neg p_{x,j,t} \wedge p_{x,j,(t+1)}\right) \rightarrow \\ \left(\bigwedge_{i \in P_j^{\varphi}} \bigvee_{y=1, y \neq x}^{k} p_{y,i,t} \right) \wedge \left(\bigwedge_{i=1}^{n} \bigwedge_{y=1, y \neq x}^{k} \neg \left(\neg p_{y,i,t} \wedge p_{y,i,(t+1)}\right) \right) \Bigg)$$

The sets A_φ and P_j^φ are to be understood as sets of indices of the respective nodes.

This encoding is polynomial, both in n and k. However constraint 4 accounts to $O(n^3 * k^2)$ clauses. Even small resolution proofs have more than 1000 nodes and pebble numbers bigger than 100, which adds up to 10^{13} clauses for constraint 4 alone. Therefore, although theoretically possible to play the pebbling game via SAT-solving, this is practically infeasible for compressing proof space. The following theorem proves the correctness of the encoding.

Theorem 4.2.1 (Correctness of pebbling SAT encoding). *$\Psi = \Psi_1 \wedge \Psi_2 \wedge \Psi_3 \wedge \Psi_4$ is satisfiable* iff *there exists a pebbling strategy using no more than k pebbles*

Proof. Suppose Ψ *is satisfiable* and let $\mathcal{I}$ be a satisfying variable assignment in form of the set of true variables. We will use $P(x, j, t)$ as an abbreviation for $p_{x,j,(t-1)} \notin \mathcal{I}$ and $p_{x,j,t} \in \mathcal{I}$. Since $\mathcal{I}$ satisfies Ψ_3, in $P(x, j, t)$ x is uniquely defined by j and t and we can write $P(j, t)$ instead. We will prove the following assertion. For every $t \in \{1, \dots, n\}$ there exists exactly one $j \in \{1, \dots, n\}$ such that $P(j, t)$.

Ψ_1 states that the root v_n has to be pebbled in the last round and Ψ_2 states that no node is pebbled initially. So for n there has to be a $t \in \{1, \dots, n\}$ such that $P(n, t)$. $\mathcal{I}$ satisfies Ψ_4, therefore for every predecessor of v_j of v_n there exists $x \in \{1,\ldots,k\}$ such that $p_{x,j,(t-1)}$. Using the same argument for v_j as for v_n there has to be a $t' \in \{1,\ldots,(t-1)\}$ such that $P(j,t')$. Every node of the proof is a recursive ancestor of the root, therefore for every $j \in \{1,\ldots,n\}$ there exists at least one $t \in \{1,\ldots,n\}$ such that $P(n,t)$. For every $t \in \{1,\ldots,n\}$, Ψ_4 ensures that if $P(n,t)$ then there is no $i \in \{1,\ldots,n\}, i \neq j$ such that $P(i,t)$, which proves the assertion. The assertion implies the existence of a bijection $\tau : \{1,\ldots,n\} \rightarrow \{v_1,\ldots,v_n\}$ such that $\tau(n) = v_n$ and $\tau(t) = j$ *iff* $P(j,t)$. Therefore $\sigma := \{\tau(1),\ldots,\tau(n)\}$ is well defined. σ is a pebbling strategy, because $\tau(n) = v_n$, rule 1 is obeyed because of Ψ_4, rule 2 is obeyed, because unpebbling moves are given implicitly (see Theorem **??**) and rule 3 is obeyed because τ is a bijection. Ψ_3 being satisfied ensures that σ uses no more than k pebbles.

Suppose there is a pebbling strategy σ using no more than k pebbles. Let the function $\mathrm{free} : \{1,\ldots,n\} \rightarrow 2^{\{1,\ldots,k\}} \setminus \emptyset$ be defined recursively as follows and $\mathrm{peb}(t) = \min(\mathrm{free}(t))$.

$$\mathrm{free}(t) = \begin{cases} \{1,\ldots,k\} & : t = 1 \\ \mathrm{free}(t-1) \setminus \{\mathrm{peb}(t-1)\} \quad \cup & : otherwise \\ \Big\{ \mathrm{peb}(s) \mid \sigma_s \in P^{\varphi}_{\sigma_{t-1}}, s \in \{1,\ldots,t-2\} \text{ and for all } v \in C^{\varphi}_{\sigma_s} & \\ \text{there exists } r \in \{1,\ldots,t-1\} : \sigma_r = v \Big\} & \end{cases}$$

Intuitively, $\mathrm{free}(.)$ keeps track of the unused pebbles in each round. If a pebble is placed on a node, it is not free anymore. Pebbles are made free again by unpebbling moves, which correspond to the second set in the recursive definition of $\mathrm{free}(.)$. Since σ uses no more than k pebbles, $\mathrm{free}(.)$ is well defined.

Let $\mathcal{I}$ be a set of variables of Ψ defined as follows. $p_{x,j,t} \in \mathcal{I}$ *iff* $t > 0$ and there exists $s \in \{1,\ldots,t\}$ such that $\mathrm{peb}(s) = x, \sigma_s = v_j$ and for all $r \in \{s+1,\ldots,t\} : x \notin \mathrm{free}(r)$.

$\mathcal{I}$ is a satisfying assignment for Ψ. Ψ_1 is satisfied, because $\sigma_n = v_n$, therefore trivially $p_{\mathrm{peb}(n),n,n} \in \mathcal{I}$. Clearly Ψ_2 is satisfied by $\mathcal{I}$ as no variables with $t = 0$ are included in $\mathcal{I}$. To see that Ψ_3 is satisfied, suppose there exist x, t, i, j such that $i \neq j$ and $\{p_{x,j,t}, p_{x,i,t}\} \subseteq \mathcal{I}$. Then by definition of $\mathcal{I}$ there exist unique t_1 and t_2 such that $\mathrm{peb}(t_1) = x, \sigma_{t_1} = v_j$ and $\mathrm{peb}(t_2) = x, \sigma_{t_2} = v_i$. From $i \neq j$ follows $v_i \neq v_j$, therefore $t_1 \neq t_2$ w.l.o.g. suppose $t_1 > t_2$. From $\mathrm{peb}(t_2) = x$, $p_{x,i,t} \in \mathcal{I}$ and $t \geq t_1 > t_2$ follows $x \notin \mathrm{free}(t_1)$, which is a contradiction to $\mathrm{peb}(t_1) = x$. Let $P(x,j,t)$ be defined as above. Then from $P(x,j,t)$ follows $\mathrm{peb}(t) = x$ and $\sigma_t = v_j$. Rule 1 of the Bounded Pebbling Game ensures that there if $P(x,j,t)$ is true, then there exists a $y\{1,\ldots,k\} \setminus \{x\}$ such that $p_{y,i,t-1} \in \mathcal{I}$. Suppose $P(x,j,t)$ and $P(y,i,t)$ both hold for some t, $x \neq y$ and $i \neq j$, then $y = \mathrm{peb}(t) = x$ and $v_j = \sigma_t = v_i$ are both contradictions. Therefore also Ψ_4 is satisfied by $\mathcal{I}$. □ □

4.3 Greedy Pebbling Algorithms

Theorem **??** and the remarks in the end of section 4.2 indicate that obtaining an optimal topological order either by enumerating topological orders or by encoding the problem as a satisfiability problem is impractical. This section presents two greedy algorithms that aim at finding good

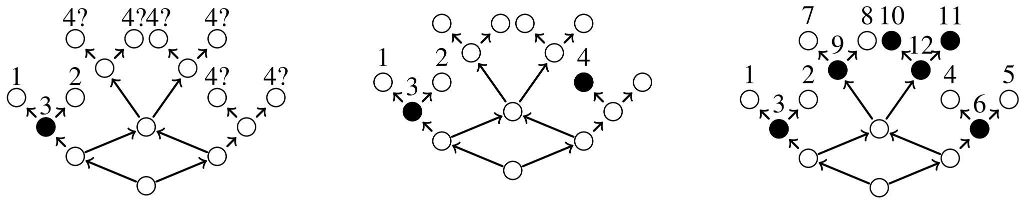

Figure 4.1: Top-Down Pebbling

though not necessarily optimal topological orders. They are both parameterized by some heuristic described in Section 4.4, but differ in the traversal direction in which the algorithms operate on proofs.

Top-Down Pebbling

Top-Down Pebbling (Algorithm 4.1) constructs a topological order of a proof φ by traversing it from its axioms to its root node. This approach closely corresponds to how a human would play the Bounded Pebbling Game. A human would look at the nodes that are available for pebbling in the current round of the game, choose one of them to pebble and remove pebbles if possible. Similarly the algorithm keeps track of pebblable nodes in a set N, initialized as A_φ. When a node v is pebbled, it is removed from N and added to the sequence representing the topological order. The children of v that become pebbleable are added to N. When N becomes empty, all nodes have been pebbled once and a topological order has been found.

Algorithm 4.1: `Top-Down Pebbling`

Input: proof φ
Output: sequence of nodes S representing a topological order $\prec$ of φ

$S = ()$; `// the empty sequence`
$N = A_\varphi$; `// initialize pebbleable nodes with Axioms`
while *N is not empty* **do**
  choose $v \in N$ heuristically;
  $S = S ::: (v)$; `// ::: is the concatenation of sequences`
  $N = N \setminus \{v\}$;
  for each $c \in C_v^\varphi$ **do** `// check whether c is now pebbleable`
    if $\forall p \in P_c^\varphi : p \in S$ **then**
      $N = N \cup \{c\}$;
return S;

Unfortunately Top-Down Pebbling often ends up finding a sub-optimal pebbling strategy regardless of the heuristic used. The following example shows such a situation.

Example 4.3.1. Consider the graph shown in Figure 4.1 and suppose that top-down pebbling has already pebbled the initial sequence of nodes $(1, 2, 3)$. For a greedy heuristic that only has information about pebbled nodes, their premises and children, all nodes marked with $4?$ are considered equally worthy to pebble next. Suppose the node marked with 4 in the middle graph is chosen to be pebbled next. Subsequently, pebbling 5 opens up the possibility to remove a pebble after the next move, which is to pebble 6. After that only the middle subgraph has to be pebbled. No matter in which order this is done, the strategy will use six pebbles at some point. One example sequence and the point where six pebbles are used are shown in the rightmost picture in Figure 4.1. However the pebbling number of this proof is 5.

Bottom-Up Pebbling

Bottom-Up Pebbling (Algorithm 4.2) constructs a topological order of a proof φ while traversing it from its root node r to its axioms. The algorithm constructs the order by visiting nodes and their premises recursively. For every node v the order in which the premises of v are visited is decided heuristically. After visiting the premises, n is added to the current sequence of nodes. Since axioms do not have any premises, there is no recursive call for axioms and these nodes are simply added to the sequence. The recursion is started with the call $BUpebble(\varphi, r, \emptyset, ())$. Since all proof nodes are ancestors of the root, the recursive calls will eventually visit all nodes once and a topological total order will be found. Bottom-Up Pebbling corresponds to the apply function $\mathrm{ap}(.)$ defined in Section **??** with the addition of a visit order of the premises. Also previously visited nodes are not visited again.

Algorithm 4.2: BUpebble

Input: proof φ
Input: node v
Input: set of visited nodes V
Input: initial sequence of nodes S
Output: sequence of nodes

$V_1 = V \cup \{v\}$;
$N = P_v^\varphi \setminus V$; `// Only unprocessed premises are visited`
$S_1 = S$;
while N *is not empty* **do**
    choose $p \in N$ heuristically; $N = N \setminus p$;
    $S_1 = S_1 ::: BUpebble(\varphi, p, V, S)$; `// ::: is the concatenation of sequences`
return $S_1 ::: (v)$;

Example 4.3.2. Figure 4.2 shows part of an execution of Bottom-Up Pebbling on the same proof as presented in Figure 4.1. Nodes chosen by the heuristic, to be processed before the respective other premise, are marked dashed. Suppose that similarly to the Top-Down Pebbling scenario, nodes have been chosen in such a way that the initial pebbling sequence is $(1, 2, 3)$. However, the choice of where to go next is predefined by the dashed nodes. Consider the dashed child of node 3. Since 3 has been completely processed, the other premise of its dashed child is visited next. The result is that the middle subgraph is pebbled while only one external node is pebbled,

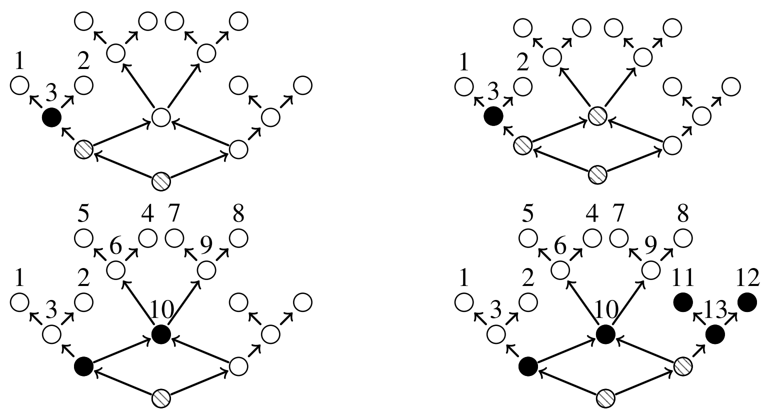

Figure 4.2: Bottom-Up Pebbling

while it were two in the Top-Down scenario. At no point more than five pebbles will be used for pebbling the root node, which is shown in the bottom right picture of the figure. This is independently of the heuristic choices.

Remarks about Top-Down and Bottom-Up Pebbling

Every topological order of a given proof can be constructed using Top-down or Bottom-up Pebbling. A heuristic that orders nodes according to the desired topological order achieves this goal. Of course such a heuristic is not very useful in practice, as we do not know the desired topological order beforehand. Both algorithms traverse the proof only once and have linear run-time in the proof length (assuming that the heuristic choice requires constant time). Therefore both algorithms are theoretically equally good in constructing topological orders.

The experiments presented in Section 4.5 show that in practice, Bottom-Up Pebbling performs much better. Example 4.3.1 shows two principles that result in pebbling strategies with small pebbling numbers and are likely to be violated by the Top-Down Pebbling algorithm.

Firstly, pebbling strategies should make local choices. By local choices we mean that it should pebble nodes that are close w.r.t. undirected edges in the graph to other pebbled nodes. Such local choices allow to unpebble other nodes earlier and therefore keep the pebbling number low. Bottom-Up Pebbling makes local choices by construction, because premises are queued up and the second premise is visited as soon as possible. Top-Down Pebbling does not have knowledge about the recursive structure of children nodes, therefore it is hard to make local choices. The algorithm simply does not know which pebbleable nodes are close to other pebbled ones.

Secondly, pebbling strategies should pebble subproofs with a high pebbling number early. Pebbling such subproofs late will result in other pebbles staying on nodes for a high number of rounds. This likely results in increasing the overall pebbling number, as this adds extra pebbles to the already high pebbling number of the subproof. The principle is more subtle than the first one, because pebbling one subproof can influence the number of pebbles used for another subproof in situations where nodes are share between subproofs. The principle is demonstrated in the following example.

Example 4.3.3. Figure 4.3 shows a simple proof φ with two subproofs φ_0 (left branch) and φ_1 (right branch). As shown in the leftmost diagram, assume $s(\varphi_0, \prec_0) = 4$ and $s(\varphi_1, \prec_1) = 5$, where $\prec_0$ and $\prec_1$ represent some topological order of the respective subproofs with the corresponding pebbling numbers. After pebbling one of the subproofs, the pebble on its root node has to be kept there until the root of the other subproof is also pebbled. Only then the root node can be pebbled. Therefore, $s(\varphi, \prec) = s(\varphi_j, \prec_j) + 1$ where $\prec$ is obtained by first pebbling according to $\prec_j$, then by $\prec_{1-j}$ followed by pebbling the root. Choosing to pebble the less spacious subproof φ_0 first results in $s(\varphi, \prec) = 6$, while pebbling the more spacious one first gives $s(\varphi, \prec) = 5$.

Note that this example shows a simplified situation. The two subproofs do not share nodes. Pebbling one of them does not influence the pebbling number of the other.

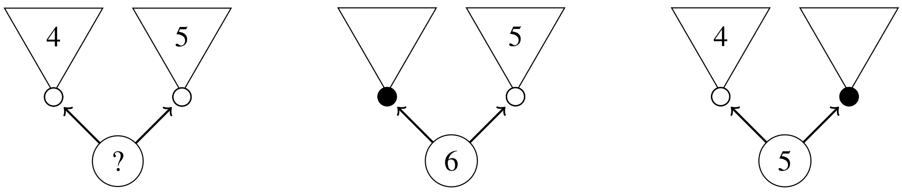

Figure 4.3: Spacious subproof first

4.4 Heuristics

Heuristics are used in both pebbling algorithms to choose one node out of a set N. For `Top-Down` Pebbling, N is the set of pebbleable nodes, and for `Bottom-Up` Pebbling, N is the set of unprocessed premises of a node.

Definition 4.4.1 (Heuristic and Full Heuristics)**.** Let φ be a proof with nodes V. A *heuristic* h for φ is a totally ordered set S_h together with a *node evaluation* function $e_h : V \to S_h$. A *full heuristic* for φ is finite a sequence $(e_{h_1}, \ldots, e_{h_n})$ of heuristics such that the node evaluation e_{h_n} is injective. The *choice* of the full heuristic for a set $N \subseteq V$ is some $v \in N$ such that $v = argmax_{v \in N} e_{h_1}(v)$ if v is unique and the choice of the full heuristic $(e_{h_2}, \ldots, e_{h_n})$ for $\{v \in N \mid v = argmax_{v \in N} e_{h_1}(v)\}$. This process will eventually terminate, because of the limitation to e_{h_n}.

Note that to satisfy the requirement for e_{h_n}, some trivial node evaluation like mapping nodes to their address in memory can be used. In the next chapters we present heuristics, which are cheap to compute and are justified by relating them to the semantics of the Bounded Pebbling Game. We will not elaborate on effects of reordering the heuristics within full heuristics.

Number of Children Heuristic (“Ch”)

The `Number of Children` heuristic uses the number of children of a node v as evaluation function, i.e. $e_h(v) = |C_v^\varphi|$ and $S_h = \mathbb{N}$. The intuitive motivation for this heuristic is that nodes with many children will require many pebbles, and subproofs containing nodes with many children will tend to be more spacious. Example 4.3.3 shows the idea behind pebbling spacious subproofs early.

Last Child Heuristic (“Lc”)

As discussed in Section **??** in the proof of Theorem **??**, the best moment to unpebble a node v is as soon as its last child w.r.t. a topological order $\prec$ is pebbled. This insight is used for the `Last Child` heuristic that chooses nodes that are last children of other nodes. Pebbling a node that allows another one to be unpebbled is always a good move. The current number of used pebbles (after pebbling the node and unpebbling one of its premises) does not increase. It might even decrease, if more than one premise can be unpebbled. For determining the number of premises of which a node is the last child, the proof has to be traversed once, using some topological order $\prec$. Before the traversal, $e_h(v) = 0$ for every node v. During the traversal $e_h(v)$ is incremented by 1, if v is the last child of the currently processed node w.r.t. $\prec$. For this heuristic $S_h = \mathbb{N}$. To some extent, this heuristic is paradoxical: v may be the last child of a node v' according to $\prec$, but pebbling it early may result in another topological order $\prec^*$ according to which v is not the last child of v'. Nevertheless, sometimes the proof structure ensures that some nodes are the last child of another node irrespective of the topological order. An example is shown in Figure 4.4, where the dashed line denotes a recursive predecessor relationship and the bottommost node is the last child of the top right node in every topological order.

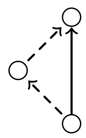

Figure 4.4: Bottommost node as necessary last child of right topmost node

Node Distance Heuristic (“$Dist(r)$”)

In Example 4.3.1 and Section 7 it has been noted that `Top-Down` Pebbling may perform badly if nodes that are far apart are selected by the heuristic. The Node `Distance` heuristic prefers to pebble nodes that are close to pebbled nodes. It does this by calculating spheres with a radius up to the parameter r around nodes. A sphere $K_r^G(v)$ with radius r around the node v in the graph $G = (V, E)$ is the set $\{p \in V \mid v \text{ can be reached from } p \text{ visiting at most } r \text{ edges}\}$, where edges are considered undirected. The heuristic uses the following functions based on the spheres:

$$
\begin{aligned}
d(v) &:= \begin{cases} -min(D) \text{ such that } D = \{r \mid K_r^G(v) \text{ contains a pebbled node}\} \neq \emptyset \\ \infty \text{ otherwise} \end{cases} \\
s(v) &:= |K_{-d(v)}^G(v)| \\
l(v) &:= max_{\prec} K_{-d(v)}^G(v) \\
e_h(v) &:= (d(v), s(v), l(v))
\end{aligned}
$$

where $\prec$ denotes the order of previously pebbled nodes. So $S_h = \mathbb{Z} \times \mathbb{N} \times P$ together with the lexicographic order using, respectively, the natural smaller relation $<$ on $\mathbb{Z}$ and $\mathbb{N}$ and $\prec$ on N. The spheres $K_r(v)$ can grow exponentially in r. Therefore the maximum radius has to be kept small.

Decay Heuristics ("$Dc(h_u, \gamma, d, com)$")

`Decay` heuristics denote a family of meta heuristics. The idea is to not only use the evaluation of a single node, but also to include the evaluations of its premises. Such a heuristic has four parameters: an underlying heuristic h_u defined by an evaluation function e_u together with a well ordered set S_u, a decay factor $\gamma \in \mathbb{R}^+ \cup \{0\}$, a recursion depth $d \in \mathbb{N}$ and a combining function $com : S_u^n \to S_u$ for $n \in \mathbb{N}$. The resulting heuristic node evaluation function e_h is defined with the help of the recursive function rec:

$$
\begin{aligned}
rec(v, 0) &:= e_u(v) \\
rec(v, k) &:= e_u(v) + com(rec(p_1, k-1), \ldots, rec(p_n, k-1)) * \gamma \\
&\quad\text{where } P_v^{\varphi} = \{p_1, \ldots, p_n\} \\
e_h(v) &:= rec(v, d)
\end{aligned}
$$

4.5 Experiments

All the pebbling algorithms and heuristics described in the previous sections have been implemented in the hybrid functional and object-oriented programming language Scala (`www.scala-lang.org`) as part of the Skeptik library for proof compression (`github.com/Paradoxika/Skeptik`) [30].

To evaluate the algorithms and heuristics, experiments were executed[1] on four disjoint sets of proof benchmarks (Table 4.1). TraceCheck_1 and TraceCheck_2 contain proofs produced by the SAT-solver `PicoSAT` [7] on unsatisfiable benchmarks from the SATLIB (`www.satlib.org/benchm.html`) library. The proofs[2] are in the TraceCheck proof format, which is one of the three formats accepted at the *Certified Unsat* track of the SAT-Competition. veriT_1 and veriT_2 contain proofs produced by the SMT-solver VeriT (`www.verit-solver.org`) on

[1] The Vienna Scientific Cluster VSC-2 (`http://vsc.ac.at/`) was used.

[2] SAT proofs: `www.logic.at/people/bruno/Experiments/2014/Pebbling/tc-proofs.zip`

Name	**Number of proofs**	**Maximum length**	**Average length**
TraceCheck_1	2239	90756	5423
TraceCheck_2	215	1768249	268863
veriT_1	4187	2241042	103162
veriT_2	914	120075	5391

Table 4.1: Proof benchmark sets

Algorithm Heuristic	**Relative Performance** (%)	**Speed** (nodes/ms)
Bottom-Up		
Children	17.52	**88.6**
LastChild	**26.31**	84.5
Distance(1)	9.46	21.2
Distance(3)	-0.40	0.5
Top-Down		
Children	-27.47	0.3
LastChild	-31.98	1.9
Distance(1)	-70.14	0.6
Distance(3)	**-74.33**	**0.1**

Table 4.2: Experimental results

unsatisfiable problems from the SMT-Lib (`www.smtlib.org`). These proofs[3] are in a proof format that resembles SMT-Lib's problem format and they were translated into pure resolution proofs by considering every non-resolution inference as an axiom.

Table 4.2 summarizes the results of the experiments. The two presented algorithms are tested in combination with the four presented heuristics. The Children and LastChild heuristics were tested on all four benchmark sets. The Distance and Decay heuristics were tested on the sets TraceCheck_2 and veriT_2. The relative performance is calculated according to Formula 4.1, where f is an algorithm with a heuristic, P is the set of proofs the heuristic was tested on and G are all combinations of algorithms and heuristics that were tested on P. The time used to construct orders is measured in processed nodes per millisecond. Both columns show the best and worst result in boldface.

[3]SMT proofs: `www.logic.at/people/bruno/Experiments/2014/Pebbling/smt-proofs.zip`

Decay γ	**Depth** d	**Combination** com	**Performance Improvement** (%)	**Speed** (nodes/ms)
0.5	1	mean	0.50	47.7
0.5	1	maximum	0.40	47.0
0.5	7	mean	0.85	14.0
0.5	7	maximum	0.76	15.3
3	1	mean	0.48	64.0
3	1	maximum	0.43	**64.4**
3	7	mean	0.21	15.3
3	7	maximum	**0.94**	15.3

Table 4.3: Improvement of LastChild using Decay Heuristic

$$\text{relative\_performance}(f, P, G) = \frac{1}{|P|} * \sum_{\varphi \in P} \left(1 - \frac{s(\varphi, f(\varphi))}{avg_{g \in G} s(\varphi, g(\varphi))} \right) \tag{4.1}$$

Table 4.2 shows that the Bottom-Up algorithm constructs topological orders with much smaller space measures than the Top-Down algorithm. This fact is visualized in Figure 4.5, where each point represents a proof φ. The x and y coordinates are the smallest space measure among all heuristics obtained for φ using, respectively, the Top-Down and Bottom-Up algorithm. The results for Top-Down range far beyond 15000, but to display the discrepancy between the two algorithms the plot scales from 0 to 15000 on both axis. The biggest best space measure for Top-Down is 131 451, whereas this number is 11 520 for the Bottom-Up algorithm. The LastChild heuristic produces the best results and the Children heuristic also performs well. The Distance heuristic produces the worst results, which could be due to the fact that the radius is too small for big proofs with thousands of proof nodes.

Table 4.3 summarizes results of the Decay Heuristic with the best results highlighted in boldface. Decay Heuristics were tested with the Bottom-Up algorithm, using LastChild as underlying heuristic. For the parameters decay factor, recursion depth and combining function two values and all their combinations have been tested. The performance improvement is calculated using Formula 4.1 with G being the singleton set of the Bottom-Up algorithm with the LastChild heuristic. The results show, that Decay Heuristics can improve the result, but not by a landslide. The improvement comes at the cost of slower speed, especially when the recursion depth is big.

Some additional heuristics, not described in this work, designed specifically for Top-Down Pebbling were tested on small benchmark sets. These heuristics aimed at doing local pebbling without having to calculate full spheres. For example pebbling nodes that allow other nodes to be unpebbled in the next move can be preferred. Unfortunately, none of the additional heuristics showed promising results.

chapters/pebbling/figures/td_vs_bu.pdf

Figure 4.5: Space measures of best Bottom-Up and Top-Down result

The Bottom-Up algorithm does not only produce better results, it is also much faster, as can be seen in the last column of Table 4.2. The reason probably is the number of comparisons that the algorithms make. For Bottom-Up the set N of possible choices consists of the premises of a single node only, and usually $|N| \in O(1)$ (e.g. for a binary resolution proof, $N \leq 2$ always). For Top-Down the set N is the set of currently pebbleable nodes, which can be large (e.g. for a perfect binary tree with $2n - 1$ nodes, initially $|N| = n$). Possibly for some heuristics, Top-Down algorithms could be made more efficient by using, instead of a set, an ordered sequence of pebbleable nodes together with their memorized heuristic evaluations.

Unsurprisingly the radius used for the Distance Heuristic has a severe impact on the speed, which decreases rapidly as the maximum radius increases. With radius 5, only a few small proofs were processed in a reasonable amount of time.

On average the smallest space measure of a proof is 44.1 times smaller than its length. This shows the impact that the usage of deletion information together with well constructed topological orders can have. When these techniques are used, on average 44.1 times less memory is required for storing nodes in memory while proof processing.

4.6 Conclusion

Several algorithms for compressing proofs with respect to space have been conceived. The experimental evaluation clearly shows that the so-called Bottom-Up algorithms are faster and compress more than the more natural, straightforward and simple Top-Down algorithms. Both kinds of algorithms are parameterized by a heuristic function for selecting nodes. The best performances are achieved with the simplest heuristics (i.e. Last Child and Number of Children). More sophisticated heuristics provided little extra compression but cost a high price in execution time. Future work could investigate heuristics that take advantage of the particular shape of proofs generated by analysis of conflict graphs.

Acknowledgments: We would like to thank Armin Biere for clarifying why resolution chains are not left-associative in the TraceCheck proof format.

CHAPTER 5

Conclusion

In this work we presented two methods for proof compression and their theoretical foundations. We implemented the methods and evaluated them extensively to show their usefulness. Furthermore, we investigated the complexity of the underlying problems and in the case of explanation production proved the complexity, which is a new result for complexity theory. For both methods we highlighted possible future work, which could improve the presented methods further. The key contributions of this work are two novel proof compression methods and the proof of NP-completeness of the shortest explanation decision problem.

Bibliography

[1] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In *CADE*, pages 64–78, 2000.

[2] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Linear-time reductions of resolution proofs. In *Haifa Verification Conference*, pages 114–128, 2008.

[3] Henk Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(02):181–215, 1997.

[4] Richard Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[5] Eli Ben-Sasson. Size space tradeoffs for resolution. In *STOC*, pages 457–464, 2002.

[6] Eli Ben-Sasson and Jakob Nordström. Short proofs may be spacious: An optimal separation of space and length in resolution. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:2, 2009.

[7] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[8] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[9] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In *CADE*, pages 151–156, 2009.

[10] Siu Man Chan. Pebble games and complexity. 2013.

[11] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Math. Program.*, 73:129–174, 1996.

[12] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, pages 345–363, 1936.

[13] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

[14] Haskell B. Curry. *Combinatory Logic*. Amsterdam, North-Holland Pub. Co., 1958.

[15] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[16] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.

[17] Juan Luis Esteban and Jacobo Torán. Space bounds for resolution. *Inf. Comput.*, 171(1):84–97, 2001.

[18] Pascal Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, PhD thesis, Institut Montefiore, Université de Liege, Belgium, 2004.

[19] Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Compression of propositional resolution proofs via partial regularization. In *CADE*, pages 237–251, 2011.

[20] Lester Randolph Ford. Network flow theory. 1956.

[21] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. *SIAM J. Comput.*, 9(3):513–524, 1980.

[22] Philipp Hertel and Toniann Pitassi. Black-white pebbling is pspace-complete. *Electronic Colloquium on Computational Complexity (ECCC)*, 14(044), 2007.

[23] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. *CoRR*, abs/1308.4767, 2013.

[24] Boris Konev and Alexei Lisitsa. A sat attack on the erdos discrepancy conjecture. *CoRR*, abs/1402.2184, 2014.

[25] Kenneth L. McMillan. Applications of craig interpolants in model checking. In *TACAS*, pages 1–12, 2005.

[26] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[27] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *RTA*, pages 453–468, 2005.

[28] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007.

[29] Jakob Nordström. Narrow proofs may be spacious: Separating space and width in resolution. *SIAM J. Comput.*, 39(1):59–121, 2009.

[30] Bruno Woltzenlogel Paleo, Joseph Boudou, and Andreas Fellner. Skeptik system description.

[31] Nicholas Pippenger. Comparative schematology and pebbling with auxiliary pushdowns (preliminary version). In *STOC*, pages 351–356, 1980.

[32] Nicholas Pippenger. *Advances in pebbling*. Springer, 1982.

[33] Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92(3):305–316, 1924.

[34] Ravi Sethi. Complete register allocation problems. *SIAM J. Comput.*, 4(3):226–248, 1975.

[35] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978.

[36] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[37] Peter van Emde Boas and Jan van Leeuwen. Move rules and trade-offs in the pebble game. In *Theoretical Computer Science*, pages 101–112, 1979.

[38] S. A. Walker and H. Raymond Strong. Characterizations of flowchartable recursions. *J. Comput. Syst. Sci.*, 7(4):404–447, 1973.