Int J Softw Tools Technol Transfer (2019) 21:71–86
https://doi.org/10.1007/s10009-017-0459-0

REGULAR PAPER

Greedy pebbling for proof space compression

Andreas Fellner[1,2] · **Bruno Woltzenlogel Paleo**[2,3]

Published online: 27 June 2017
© The Author(s) 2017. This article is an open access publication

Abstract Automated reasoning tools for the verification and synthesis of software often produce proofs to allow independent certification of the correctness of the produced solutions. As proofs can be large, this paper considers the problem of compressing proofs with respect to their *space*, which is approximately proportional to the memory necessary to check them. Proof checking with a small amount of available memory is analogous to playing a *pebbling game* with a small number of pebbles. This paper exploits this analogy and describes novel algorithms for playing a *pebbling game*. The sequence of moves executed in the pebbling game then corresponds to an improved topological ordering of the nodes of the proof, leading to smaller memory consumption when the proof is checked. Because the number of possible pebbling strategies and topological orderings is too large, brute-force approaches to find optimal solutions are impractical, and hence, the new pebbling algorithms proposed here are based on heuristics for finding good, though not necessarily optimal, solutions. The algorithms are evaluated on the task of compressing the space of thousands of propositional resolution proofs generated by SAT- and SMT-solvers.

Keywords Proof compression · Memory consumption · Resolution · Pebbling games

✉ Andreas Fellner
fellner.a@gmail.com; afellner@forsyte.tuwien.ac.at

Bruno Woltzenlogel Paleo
bruno@logic.at

[1] Austrian Insitute of Technology, Vienna, Austria

[2] Vienna University of Technology, Vienna, Austria

[3] Australian National University, Canberra, Australia

1 Introduction

Proofs have the potential to play an important role in ensuring trust between independent software tools that transfer knowledge, solutions and computation results among each other, because proofs can serve as certificates of correctness of what is communicated. This potential is already being realized in the field of automated reasoning, where proof exchange for theorem proving [11,40] is routinely used to enable verified cooperation among a vast variety of proof-producing automated deduction tools [66], as exemplified in systems such as Sledgehammer [50] and HOL(y)Hammer [41].

SAT- and SMT-solvers are widely used as subroutines to solve problems in various domains. Clearly, solvers are not immune to bugs [15,16] that may lead to wrong results. Fortunately, there is a simple way to certify the output of such solvers and prevent the potentially dangerous propagation of an error in them.

In case of the positive result "satisfiable formula," the certificate is a model of the formula. Checking a model for spuriousness can be done by substituting the variables of the formula with the respective values in the model and calculating the truth value of the resulting variable-free formula.

In case of the negative result "unsatisfiable formula," the certificate is a proof of unsatisfiability. Checking a proof for spuriousness can be done by replaying the proof and validating the correct application of every proof rule. Typically, this is a much more expensive task than checking whether a model is spurious. Nevertheless, to encourage certification of unsatisfiability, the yearly SAT-competition[1] has introduced a Certified UNSAT track in 2013.

An example of an application where certified correctness is crucial is the verification and synthesis of software [10,

[1] http://baldur.iti.kit.edu/sat-competition-2016/.

38]. In an ongoing project for interpolant-based controller synthesis [38], for example, extracting an interpolant from an SMT-proof took hours and reached the limit of memory (256 GB) available in a single node of the computer cluster used in the project. The example shows one serious issue with proofs: They tend to be huge, easily filling up all available memory, and therefore are hard to process independently.

Even when a single proof is not large enough to consume all available memory, it is still important to reduce memory consumption as much as possible. In many scenarios, such as *proof-carrying code* [47] and *foundational proof certificates* [20,45], it is advocated that proofs should be made more pervasive and universal [22]. A proof checker (or a proof consumer, more generally), in these scenarios, will not be an isolated application processing a single proof on a dedicated machine with a large amount of resources. Instead, it may be running on a machine with significantly less resources than the machine used to produce the proof, sharing the limited resources with many other applications running on the same machine and having to process many proofs in parallel. In fact, in proof assistants like Isabelle [48], this is already observed to a limited extent today. Large libraries contain many lemmas with relatively small proofs that need to be re-checked and reconstructed. The smaller the memory footprint of processing a proof, no matter how small it is, the more proofs a proof consumer will be able to process in parallel without impacting other activities of the user.

Typically, proof formats (e.g., the TraceCheck format [9] for propositional resolution proofs used by SAT-solvers and formats for SMT-proofs [5]) do not allow proof producers to inform the proof consumer when proof nodes (containing clauses) could be released from memory. Consequently, every proof node loaded into memory has to be kept there until the whole proof is completely processed, because the proof consumer does not know whether the proof node will still be needed. Two exceptions are the DRUP and DRAT formats [37], which introduced deletion instructions to the previous RUP format, but motivated by different issues. For very large problems, a SAT-solver may not have enough memory to store the resolution proof during the search, the performance overhead associated with the bookkeeping of proofs in memory may be unacceptable, or optimized conflict analysis and in-processing techniques may not be easily expressible in terms of resolution. The RUP format solves these issues by allowing the solver to output only the derived clauses, without information about the DAG structure of the proof and about the premises used to derive the clauses. Furthermore, in the RUP format, solvers typically eagerly write every derived clause to the proof file during the search, even if the clause might not turn out to be useful to derive the empty clause. Consequently, RUP proofs tend to be huge, containing many useless clauses, and time-consuming to check (typically in a bottom-up way). With the DRUP format, the SAT-solver can also write down, in the proof file, when a clause was deleted during the search. This allows the bottom-up proof checker to ignore delete clauses and be more efficient.

Although SAT-solvers are unable to output resolution proofs for very large problems or when some optimizations and in-processing techniques are used, resolution proofs (either directly generated or obtained through conversion from a DRUP/DRAT proof [37]) are preferable from a proof consumer's point of view. Resolution proofs (when kept in memory during the search and written to file only when the search is done) do not need to contain useless clauses; therefore, they are typically smaller, even though they contain additional information about premises used to derive each clause. Checking a resolution step can be done locally, by inspecting only the derived clause and its premises, whereas checking a derived clause in a DRUP/DRAT proof may require inspection of an a priori unknown number of clauses located anywhere above the derived clause in the proof file. The parsimony of the resolution calculus also makes checking a resolution proof conceptually simpler and implementable in fewer lines of trusted code. The extra information contained in resolution proofs is essential for further manipulation and compression of the proof [3,13,21,29,57] and for applications that rely, for instance, on interpolants extracted from proofs [23,38,51]. And finally, although alternative detailed proof systems for conflict-driven clause learning, mixing resolution and natural deduction have recently been proposed [60], resolution remains the primary format chosen by major SMT-solvers [4,5,14] and conversion from DRUP to resolution is possible for SAT-solvers that are not able to output resolution proofs directly [37].

This paper defines new algorithms that post-process *resolution* proofs in order to include deletion instructions that could be used by a (top-down) proof consumer to reduce the memory it needs when processing the proofs. The more deletion instructions, the less memory the proof consumer will need.

The new methods proposed here exploit an analogy between proof checking and playing *pebbling games* [31,42]. In Sect. 2, we define propositional resolution proofs and make the notion of processing a proof formal. The particular version of pebbling game relevant for proof processing is defined precisely in Sect. 3, where we also explain the analogy to proof processing in detail and define the space measure of proofs. The proposed pebbling algorithms are greedy (Sect. 5) and based on heuristics (Sect. 6). As discussed in Sects. 3 and 4, approaches based on exhaustive enumeration or on encoding as a SAT problem would not fare well in practice.

The proof space compression algorithms described here are not restricted to proofs generated by SAT- and SMT-solvers. They are general DAG pebbling algorithms that

could be applied to proofs represented in any calculus where proofs are directed acyclic graphs (including the special case of tree-like proofs) [66]. It is nevertheless in SAT and SMT that proofs tend to be largest and in most need of space compression. The underlying propositional resolution calculus satisfies the DAG requirement. The experiments (Sect. 7) evaluate the proposed algorithms on thousands of SAT- and SMT-proofs.

1.1 Related work

To the best of our knowledge, the methods presented are the first to compress proof *space*, which relates to memory consumption during proof processing. Nevertheless, there have been many works aimed at compressing proof *length* or *size*, which relate to total runtime of processing proofs, in various calculi.

In the case of sequent calculus proofs, Herbrand sequent extraction [35,62] was one of the earliest proposed techniques. It consists of obtaining a propositionally valid sequent from a first-order proof with the same propositional structure of the proof's end-sequent but with all needed instantiations for the quantified variables. This technique was later generalized to higher-order logic [44] using expansion trees [46], which is a more suitable data structure than sequents for higher-order logic. A more ambitious approach to compressing sequent calculus proofs is the introduction of cuts. As every cut inference involves the derivation of a lemma (in its left premise) and its use (in the right premise), introducing cuts requires solving the difficult task of synthesizing lemmas. The first method [63] to address this problem introduced atomic cuts by using the resolution calculus, which is based on atomic cuts. A few years later, another method [34], based on discovering a grammar that could generate the Herbrand sequent of the proof to be compressed and then constructing a proof with cuts based on that grammar, was also proposed and implemented in `GAPT` [24,26,36,44,55].

More recently, for natural deduction proofs, which are common among proof assistants (e.g., Isabelle [48], Coq [39], Matita [1], Twelf [58], Beluga [52]), the only known technique involves reproving the theorem in *Contextual Natural Deduction* [64,65], which allows at least quadratic asymptotic best-case compression. However, this technique is still limited to minimal logic only and needs to be extended to more complex logics (e.g., higher-order type systems) in order to be practically useful for the mentioned proof assistants.

It is in the case of propositional resolution, which enjoys high popularity among SAT-solvers and SMT-solvers, that proof compression algorithms have been most studied and applied. The first two algorithms for compressing propositional resolution proof length were *RecycleUnits* and *RecyclePivots* [3], with the latter consisting of making the proof more *regular* by deleting parents of irregular nodes when possible. This algorithm was later improved with the invention of *RecyclePivotsWithIntersection* [29], which discovered and used a more general way of detecting deletable irregularities. Combinations of this improved algorithm with a novel technique called *LowerUnits* [29] were also described and evaluated there. *LowerUnits* was then also improved further by postponing resolutions not only with unit clauses, but also with non-unit clauses satisfying *univalence* conditions. The resulting algorithm was called *LowerUnivalents* [13]. All these algorithms traverse the proof only a constant number of times. However, there are also algorithms that do not have this property. The most well known are *Split* [21], which splits the refutation into a proof p and a proof $\neg p$ for an heuristically chosen atom p and then recombines the two proofs by resolving p and $\neg p$ into a hopefully smaller new refutation, and *ReduceAndReconstruct* [57], which looks for local patterns of redundancy and locally rewrites them, occasionally also shuffling the order of proof nodes to expose new local patterns. Both algorithms can be iterated an unbounded number of times, and the amount of compression they achieve typically depends on how many iterations are executed.

LowerUnits has been generalized from propositional- to first-order resolution [32], which can be considered a popular foundation for first-order superposition-based automated theorem provers. The algorithms *Split* and *RecyclePivotsWithIntersection* are currently being generalized to first-order logic as well.

All natural deduction, propositional and first-order resolution proof compression algorithms mentioned have been implemented by various people in `Skeptik` [12], whereas the sequent calculus algorithms have been implemented either in the `CERes` system [25] for cut-elimination by resolution, or in its successor `GAPT` [24,26,36,44,55] (the General Architecture for Proof Theory).

Although there has never been a proof compression algorithm targeting *space*, instead of *length* or *size*, theoretical studies about trade-offs between these and other measures have been investigated in the related field of propositional proof complexity (e.g., [6]). The theoretical works in that neighboring field inspired us to develop constructive methods to obtain a proof with low space, as presented here. One important subtlety is that here we consider traversal orders of a fixed proof instead of constructing a completely new proof. The motivation for this restriction is that proof compression should ultimately deliver proofs with low size and low space at the same time. We are not interested in reducing space further at the expense of increasing size or length.

This work was inspired by the deletion instructions introduced by the DRUP format, but differs from it in a few ways. Firstly, the focus here is on resolution proofs, instead of (D)RUP proofs. Secondly, whereas the main motivation for deletion information in the DRUP format was to speed

up proof checking, the main concern here is memory consumption. And finally, whereas DRUP proofs are typically processed in a bottom-up way, here it is assumed that the proof consumer will do a top-down traversal of the proof, as usual for resolution [2,19,24–26,36,44,55].

2 Propositional resolution proofs

Resolution is among the most prominent formal calculi for automated deduction and goes back to Robinson [56]. Propositional resolution can be seen as a simplification of first-order logic resolution to propositional logic. For basics about propositional logic and its prominent decision problem SAT, we refer the reader to [10]. For an extensive discussion of propositional and first-order logic resolution, we refer the reader to [43].

Definition 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 $\overline{\neg p} = p$). A *clause* is a set of literals. $\bot$ denotes the *empty clause*.

A clause represents the propositional logic formula that is the disjunction of its literals. A set of clauses represents the formula that is the conjunction of its clauses. The propositional resolution calculus derives new clauses using the propositional resolution rule with the aim of deriving the empty clause.

Definition 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 $C_1 \backslash \{\ell\} \cup C_2 \backslash \{\overline{\ell}\}$ is called the *resolvent* of C_1 and C_2 with *pivot* 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, then the clause contains both the positive and negative versions 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 ignore 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 be tautological. Therefore, we will drop the reference to the literal when speaking about resolvents.

It is usual to present proofs in this calculus as syntactic derivations and refutations that are sequences of clauses. However, in this work, we investigate the graph structure of proofs and therefore present proofs as graphs in the following definition.

Definition 3 (*Proof*) A *proof* φ is a labeled directed acyclic graph $\langle V, E, v, \mathcal{L}\rangle$, such that v is a unique root of the graph, that is, v has no incoming edges and every node is reachable from v, $\mathcal{L}$ maps nodes to clauses and one of the following properties is fulfilled:

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 $x \in V_R : \mathcal{L}(x) = \mathcal{L}_2(x), V = (V_L \cup V_R) \cup \{v\}, E = E_L \cup E_R \cup \{(v, v_L), (v, v_R)\}$.

For a node $v \in V$, $\mathcal{L}(v)$ is the *conclusion* of v. In case 2, v_L and v_R are *premises* of v and v is a *child* of v_L and v_R. A proof ψ is a *subproof* of a proof φ, if the respective roots are related in the transitive closure of the premise relation. The root of a subproof ψ of φ which has no premises is an *axiom* of φ. A_φ denotes the set of axioms of φ. P_v^φ denotes the premises and C_v^φ the children of a node v in a proof φ.

Note that since the labeling of premises must agree on common nodes and edges, the definition of the labeling $\mathcal{L}$ is unambiguous. Also note that in case 2 of Definition 3, V_L and V_R are not required to be disjoint. Therefore, the underlying structure of a proof is really a directed acyclic graph and not simply a tree. Modern SAT- and SMT-solvers, using techniques of conflict-driven clause learning, produce proofs with a DAG structure [10,14]. The reuse of proof nodes plays a central role in proof compression [29].

Several measures can be defined on proofs. The relevant measure for this work is space, which is defined in Sect. 3. Other common measures of proofs that are not discussed in this work are, for example, length, height, width and size of the UNSAT core.

Example 1 Consider the propositional logic formula Φ, displayed in clause notation.

$$\Phi := \langle \{x_1, x_2, \neg x_3\}, \{x_1, \neg x_2\}, \{x_1, x_3\}, \{\neg x_1\}\rangle$$

By resolving the clauses $\{x_1, x_2, \neg x_3\}$ and $\{x_1, \neg x_2\}$, 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 $\bot$ by resolving $\{x_1\}$ with $\{\neg x_1\}$, which proves that Φ is unsatisfiable. The resulting proof is displayed in Fig. 1.

The aim of this work is to make proof processing easier by minimizing space requirements of proofs. Proof processing could be checking its correctness, manipulating it, as we do in this work extensively, or extracting information, for example interpolants and UNSAT cores, from it. The following definition makes the notion of proof processing formal.

Definition 4 (*Proof processing*) Let $\varphi = \langle V, E, v, \mathcal{L}\rangle$ be a proof and T be an arbitrary set. A function $f : V \times T \times T \rightarrow$

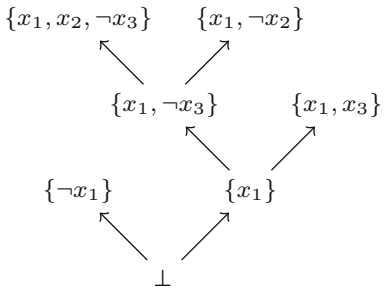

Fig. 1 Proof of Φ's unsatisfiability

T is a *processing function* if there is a function $g_f : V \to T$ such that for every $v \in A_\varphi : g_f(v) = f(v, t_1, t_2)$ for all $t_1, t_2 \in T$. Let $\mathcal{F}$ be the set of processing functions. The *apply function* $\alpha : V \times \mathcal{F} \to T$ is defined recursively as follows.

$$\alpha(v, f) = \begin{cases} f(v, \alpha(p_1, f), \alpha(p_2, f)) & \text{if } P_v^\varphi = \{p_1, p_2\} \\ g_f(v) & \text{otherwise} \end{cases}$$

Processing a node v with some processing function f means computing the value $\alpha(v, f)$. *Processing a proof* means processing its root node.

The definition above describes proof processing that traverses the proof in a top-down way. Applying a function f to a node v depends on the results of previously applying f to the premises p_1 and p_2 of v, which occur above v.

Example 2 Checking the correctness of some proof (i.e., checking for the absence of faulty resolution steps) can be done in terms of the following processing function with $T = \{\top, \bot\}$ and $\wedge$ being the usual boolean AND operation.

$$f(v, w_1, w_2) = \begin{cases} \top & \text{if } v \text{ has no premises} \\ w_1 \wedge w_2 & \text{if } P_v^\varphi = \{p_1, p_2\} \text{ and} \\ & \quad \mathcal{L}(v) \text{ is a resolvent of} \\ & \quad \mathcal{L}(p_1) \text{ and } \mathcal{L}(p_2) \\ \bot & \text{otherwise} \end{cases}$$

Processing a proof with processing function f yields $\top$ if and only if the proof is all resolution steps in the proof are correct.

As indicated in the example above, proof checking is assumed to be done in a top-down way throughout this paper. This is the common way to process all kinds of proofs in proof checking systems such as `CERes` [25], `GAPT` [24,26,36,44,55], the `Checkers` tool [19] for foundational proof certificates [20] and `Dedukti` [2,17]. Moreover, when the conclusion clauses of resolution inferences (or chains of inferences) are omitted in propositional resolution proofs in the TraceCheck format, only top-down checking is possible, because the conclusion clauses need to be recomputed based on their premises.

Another example of proof processing is any proof compression algorithm that might alter the structure of a proof. In that case, the set T contains (potentially new) proof nodes.

3 Pebbling game and space

Pebbling games denote a family of games played on graphs where nodes are marked and unmarked throughout the rounds of the games. The goal of these games is to mark some designated node. On top of the number of rounds played to achieve the goal, an interesting characteristic of a particular instance of a pebbling game is the maximal amount of nodes that are marked simultaneously over the course of all rounds. The latter characteristic is the one we are interested in, because it models space requirements, when marking a node is interpreted as loading it into memory. In the context of pebbling games, it is common to use the phrase to (un)pebble a node for (un)marking it.

Pebbling games were introduced in the 1970s to model the expressive power of programming languages [53,61] and compiler construction [59]. More recently, they have been used to investigate various questions in parallel complexity [18] and proof complexity [7,28,49]. Pebbling games are used to obtain bounds for space and time requirements and trade-offs between the two measures [6,27].

There is a variety of different pebbling games that differ in the rules and how many types of pebbles are used. In the following definition, we present the pebbling game that we use to model space requirements of proofs, which uses a single kind of pebble.

Definition 5 (*Bounded pebbling game*) The *bounded pebbling game* is played by one player in rounds on a DAG $G = (V, E)$ with one distinguished node $v \in V$. The goal of the game is to pebble v, respecting the following rules:

1. A node $v \in V$ is pebbleable in a round if and only if all predecessors of v in G are pebbled in this round and v is currently not pebbled.
2. Pebbled nodes can be unpebbled in any round.
3. Once a node has been unpebbled, it may not be pebbled in a later round.

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.

We display examples of this game in Sect. 5, when we discuss algorithms to construct strategies for it.

Note 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. When playing the bounded pebbling game on a proof φ, the designated target node is its root.

Definition 6 (*Strategy*) For a bounded pebbling game, played on a DAG $G = (V, E)$ with distinguished node v, a *pebbling strategy* σ is a sequence of moves $(\sigma_1, \ldots, \sigma_n)$ of the player such that $\sigma_n = p(v)$. We denote the set of nodes that are pebbled in round i by

$$\begin{aligned}\mathrm{Peb}_i^\sigma := \{v \in V \mid \exists j \leq i : \sigma_j = p(v) \wedge \\ \forall k : j < k \leq i : \sigma_k \neq u(v)\}\end{aligned}$$

Furthermore, we denote the set of nodes that are ready to be unpebbled in round i by

$$\mathrm{UPeb}_i^\sigma := \left\{v \in \mathrm{Peb}_i^\sigma \mid \forall c \in C_v^\varphi c \in \mathrm{Peb}_i^\sigma\right\}$$

The following definition allows us to measure the amount of pebbles necessary to play the bounded pebbling game on a given graph.

Definition 7 (*Pebbling number*) The *pebbling number of a pebbling strategy* $(\sigma_1, \ldots, \sigma_n)$ is defined as the maximum number of pebbled nodes in all rounds, i.e., $max_{i \in \{1 \ldots n\}} |\mathrm{Peb}_i^\sigma|$. The *pebbling number of a DAG G and distinguished node v* is the minimum pebbling number over all pebbling strategies for G and v.

The bounded pebbling game from Definition 5 differs from the black pebbling game discussed in [33,54] in two aspects. Firstly, the black pebbling game does not include rule 3. Excluding this rule allows for pebbling strategies with lower pebbling numbers ([59] has an example on page 1), at the expense of an exponential upper bound on the number of rounds [27]. 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 [27]. Deciding whether the pebbling number of a graph G and node v is smaller than k is PSPACE-complete in the absence of rule 3 [31] and NP-complete when rule 3 is included [59].

Our interpretation of the game is that every round of the game corresponds to an I/O operation and, if the action of the player is to pebble a node, the processing of the node. The goal of proof compression is to make proof processing less expensive. Therefore, admitting exponentially many I/O operations and processing steps in the worst case is not a viable option. That is the reason why we chose the bounded pebbling game for our purpose. In the bounded pebbling game, the number of rounds is linear in the number of nodes, since every node is pebbled and unpebbled at most once.

In order to process a node according to Definition 4, the results of processing its premises are used and therefore have to be stored in memory. The requirement of having premises in memory corresponds to rule 1 of the bounded pebbling game. A node that has been processed can be removed from memory, which corresponds to rule 2. Note that removing a node and its results too early in combination with rule 3 makes it impossible to process the whole proof. The optimal moment to remove a node from memory is uniquely determined by the order that nodes are processed (see Theorem 1).

Definition 4 does not specify in which order to process nodes. The order in which nodes are processed is essential for the memory consumption, just like the order of pebbling nodes in the pebbling game is essential for the pebbling number. The following definition allows us to relate pebbling strategies with orderings of nodes.

Definition 8 (*Topological order*) A topological order of a proof φ with nodes V is a total order relation $\prec$ on V, such that for all $v \in V$, for all $p \in P_v^\varphi : p \prec v$. A sequence of moves $(\sigma_1, \ldots, \sigma_n)$ in the pebbling game *respects* a topological order $\prec$ if for all $j, i \in \{1, \ldots, n\}$ such that $\sigma_j = p(v_j)$ and $\sigma_i = p(v_i)$ it is true that $j < i$ if and only if $v_j \prec v_i$.

A topological order $\prec$ of a proof φ can be represented as a sequence $(v_1, \ldots, v_n)$ of proof nodes, by defining $\prec := \{(v_i, v_j) \mid 1 \leq i < j \leq n\}$. The requirement that topological orders rank premises lower than their children corresponds to rule 1 of the bounded pebbling game. The antisymmetry together with the fact that $V = \{v_1, \ldots, v_n\}$ corresponds to rule 3. Theorem 1 shows that the rounds for unpebbling moves are predefined by the pebbling moves, when the goal is to find strategies with small pebbling numbers. Therefore, there is a bijection between topological orders and canonical pebbling strategies.

Definition 9 (*Canonical pebbling strategy*) The *canonical pebbling strategy* σ for a proof φ, its root node s and a topological order $\prec$ represented as a sequence $(v_1, \ldots, v_n)$ is defined recursively:

$$\sigma_1 = p(v_1)$$

$$\sigma_i = \begin{cases} u(v) & \text{if } \mathrm{UPeb}_i^\sigma \neq \emptyset, \text{ where} \\ & v = \min_\prec \left(\mathrm{UPeb}_i^\sigma\right) \\ p(v) & \text{otherwise, where} \\ & v = \min_\prec (w \mid \text{ for all } l < i : \sigma_l \neq p(w)) \end{cases}$$

Intuitively, the strategy pebbles the nodes in the order in which they are given, and as soon as it is possible to unpebble a node, it does so immediately. The following theorem

shows that unpebbling moves can be omitted from strategies for the bounded pebbling game, when the goal is to produce strategies with low pebbling numbers.

Theorem 1 *The canonical pebbling strategy has the minimum pebbling number among all pebbling strategies that respect the topological order* $\prec$.

Proof Let $\sigma = (\sigma_1, \ldots, \sigma_n)$ be the canonical pebbling strategy for $\prec$ and let $\gamma = (\gamma_1, \ldots, \gamma_n)$ be any pebbling strategy respecting $\prec$.

Let $\#P_i^\delta$ and $\#U_i^\delta$ be the number of pebbling and unpebbling moves, respectively, performed by $\delta \in \{\sigma, \gamma\}$ up to round i of the game. Rule 3 of the bounded pebbling game disallows to play a pebbling move on the same node more than once. Furthermore, by our definition of unpebbling moves, such moves are only played on nodes that are pebbled in the respective round. Therefore, we can characterize the number of pebbled nodes in round i by strategy δ as $|Peb_i^\delta| = \#P_i^\delta - \#U_i^\delta$.

Pebbling and unpebbling moves are the only possible moves, and the player is forced to play a move every turn. Therefore, we have $\#P_i^\delta = i - \#U_i^\delta$ and $|Peb_i^\delta| = i - 2\#U_i^\delta$.

Strategies σ and γ respect the same topological order $\prec$. Therefore, nodes are available for unpebbling in the same sequence for the two strategies. Strategy σ prioritizes unpebbling over pebbling moves, i.e., it does the maximum amount of unpebbling moves possible. Thus for every i, we have $\#U_i^\sigma \geq \#U_i^\gamma$, which implies $|Peb_i^\sigma| \leq |Peb_i^\gamma|$. Since we have the property for every i, it also holds for the maximum over all i, which is the desired property. □

As a consequence of Theorem 1, finding pebbling strategies with low pebbling numbers can be reduced to constructing topological orders. The memory required to process a proof using some topological order can be measured by the pebbling number of the canonical pebbling strategy corresponding to the order. We are now ready to define another measure on proofs, which we call space.

Definition 10 (*Space of a proof*) The *space* $s(\varphi, \prec)$ of a proof φ and a topological order $\prec$ is the pebbling number of the canonical pebbling strategy of φ, its root and $\prec$.

Note that the space $s(\varphi, \prec)$ of a proof φ and a topological order $\prec$ is an abstract idealized approximation of the memory consumption needed by a proof consumer processing φ according to the canonical strategy corresponding to $\prec$. It is a good approximation, because the size of non-leaf nodes can be assumed to be constant, since non-tautological resolvents (cf. Definition 2) are uniquely determined by their premises and do not need to be explicitly stored in memory. Only the size of leaf nodes varies depending on the size of the input clauses they contain.

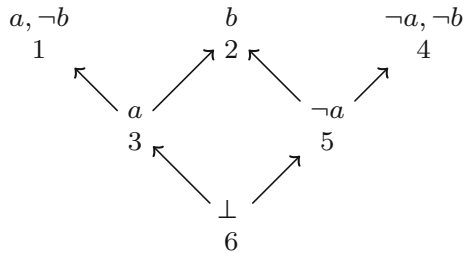

Fig. 2 A simple proof

Example 3 Consider the proof displayed in Fig. 2. The indices below the proof nodes indicate a topological order that has pebbling number four. The implicit unpebbling moves are to unpebble node 1 after pebbling node 3, as well as unpebbling nodes 2 and 4 after pebbling node 5. Before unpebbling nodes 2 and 4, nodes 2, 3, 4, 5 are pebbled which is the maximal amount of pebbles placed on the graph at any time. It is easy to see that there is no topological order that has a canonical pebbling strategy with a lower pebbling number.

The problem of compressing the space of a proof φ and a topological order $\prec$ is the problem of finding another topological order $\prec'$ such that $s(\varphi, \prec') < s(\varphi, \prec)$. The following theorem shows that the number of possible topological orders is very large and, hence, enumeration is not a feasible option when trying to find a good topological order.

Theorem 2 *There is a sequence of proofs* $(\varphi_1, \varphi_2, \ldots)$ *such that* $|T(\varphi_m)| \in \Omega(l(\varphi_m)!)$, *where* $T(\varphi_m)$ *is the set of possible topological orders for* φ_m *and* $l(\varphi)$ *denotes the number of nodes of* φ.

Proof Let φ_m be a perfect binary tree with 2^m axioms. We can calculate the length of the tree as $n := l(\varphi_m) = 2 * 2^m - 1$. Let $(v_1, \ldots, v_n)$ be a topological order for φ_m. Let $A_{\varphi_m} = \{v_{k_1}, \ldots, v_{k_{2^m}}\}$, then $(v_{k_1}, \ldots, v_{k_{2^m}}, v_{i_1}, \ldots, v_{i_{n-2^m}})$, where the indices are such that $(i_1, \ldots, i_{n-2^m}) = (1, \ldots, n) \setminus (k_1, \ldots, k_{2^m})$ is a topological order as well.

Likewise, for every permutation π of $\{k_1, \ldots, k_{2^m}\}$, $(v_{\pi(k_1)}, \ldots, v_{\pi(k_{2^m})}, v_{i_1}, \ldots, v_{i_{n-2^m}})$ is a topological order. There are $2^m!$ such permutations, so the overall number of topological orders is at least factorial in 2^m, thereby also in $n = l(\varphi_m)$. □

There might not only be many possible topological orders, their pebbling numbers might also be substantially different.

Theorem 3 *There is a sequence of proofs* $(\varphi_1, \varphi_2, \ldots)$ *such that there are topological orders* δ, γ *for* φ_m *with pebbling numbers* n_δ *and* n_γ, *such that* $n_\delta = O(2^{n_\gamma})$.

Proof Again, let φ_m be a perfect binary tree with 2^m axioms. Let $\delta = (v_{k_1}, \ldots, v_{k_{2^m}}, v_{l_1}, \ldots, v_{l_{n-2^m}})$ be the topological order, where indices are defined as in the proof of Theorem 2.

The strategy initially pebbles all axioms, making no node available for unpebbling. Only after pebbling one additional node, two axiom nodes can be unpebbled. Therefore, we have that $n_\delta = 2^m + 1$.

Let γ be the strategy that processes φ_m from left to right. We show by induction on $m : n_\gamma = m + 2$. The base case is $m = 1$. The strategy γ has to pebble both axioms first, before it is able to pebble the root node. Therefore, we have $n_\gamma = 3 = 1 + 2$ for φ_1.

Let φ_m be a perfect tree with 2^m axioms, which has a left and a right subproof, which are perfect binary trees with 2^{m-1} axioms. By induction hypothesis, we have that γ needs $(m-1)+2$ pebbles on the left and right subproofs. After processing the left subproof, one pebble remains on the root of the left subproof, while processing the right subproof. Therefore, we have $n_\gamma = (m-1) + 3 = m + 2$ for φ_m.

We have $n_\delta = 2^m + 1 = O(2^{m+2}) = O(2^{n_\gamma})$. □

4 Pebbling as a satisfiability problem

Pebbling is a graph problem, and many graph problems can be encoded as SAT problems and, at least in principle, solved by running a SAT- solver [10]. Not surprisingly, there exists a SAT encoding for the problem of deciding whether a proof can be pebbled using no more than k pebbles, and the pebble number of a proof could in principle be found by trying increasingly larger values of k. In this section, we present the SAT encoding and briefly discuss its complexity, in order to argue that finding the pebble number through SAT encodings is not a practical solution.

In this section, φ is assumed to be a proof with nodes $v_1, \ldots, v_n$ with v_n its root. Due to rule 3 of the bounded pebbling game, the number of moves that pebble nodes is exactly n, and due to Theorem 1, determining the order of these moves is enough to define a strategy.

In our SAT encoding, 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.

For pebbling strategies, it is not relevant which of the k pebbles is on a node. Therefore, one could also think of an encoding where true variables simply mean that a node is pebbled. However, such an encoding would require exponentially many clauses (in k) or a smart way to cope with cardinality constraints, when limiting the number of pebbles used in a round.

Definition 11 (*Pebbling SAT encoding*) The conjunction of the following four constraints expresses the existence of a pebbling strategy for φ with pebbling number smaller than or equal to 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

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

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.

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

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 larger 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, unless a more efficient encoding is found. The following theorem states the correctness of our encoding.

Proposition 1 (Correctness of pebbling SAT encoding) $\Psi = \Psi_1 \wedge \Psi_2 \wedge \Psi_3 \wedge \Psi_4$ *is satisfiable if and only if there exists a pebbling strategy with pebbling number smaller than or equal to* k.

5 Greedy pebbling algorithms

Theorem 2 and the remarks in the end of Sect. 4 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 though not necessarily optimal topological orders. They are both parameterized by some heuristic described in Sect. 6, but differ in the traversal direction in which the algorithms operate on proofs.

5.1 Top-down pebbling

Top-down pebbling (Algorithm 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 1: `Top-Down Pebbling`

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

```
1  S = ();                                  // the empty sequence
2  N = A_φ;                                 // pebbleable nodes
3  while N is not empty do
4      choose v ∈ N heuristically;
5      S = S ::: (v);   // ::: is sequence concatenation
6      N = N \ {v};
           // check whether c is now pebbleable
7      for each c ∈ C_v^φ do
8          if ∀p ∈ P_c^φ : p ∈ S then
9              N = N ∪ {c};
10 return S;
```

Top-down pebbling often constructs pebbling strategies with high pebbling numbers regardless of the heuristic used. The following example shows such a situation.

Example 4 Consider the graph shown in Fig. 3, 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 top-right graph is chosen to be pebbled next. Subsequently,

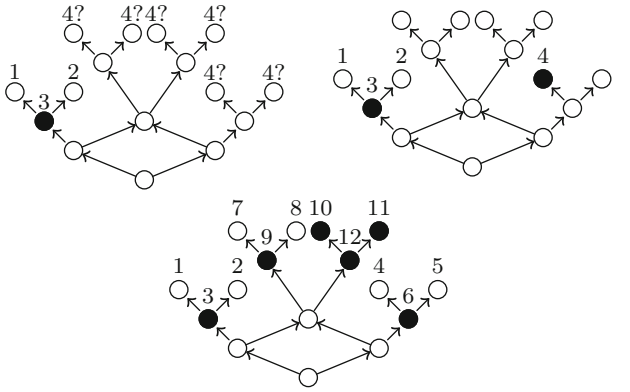

Fig. 3 Top-down pebbling

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 is shown in the bottom graph in Fig. 3. However, the pebbling number of this proof is five.

5.2 Bottom-up pebbling

Bottom-up pebbling (Algorithm 2) constructs a topological order of a proof φ while traversing it from its root node v 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, v 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 `bottom-up pebbling`$(\varphi, v, \emptyset, ())$. 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 $\alpha(.)$ defined in Sect. 2 with the addition of a visit order of the premises. Also previously visited nodes are not visited again.

Example 5 Figure 4 shows part of an execution of the bottom-up pebbling algorithm on the same proof as presented in Fig. 3. 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 with only one pebble placed on a node that does not belong to the subgraph. In the top-down scenario, there were two such external pebbles. At

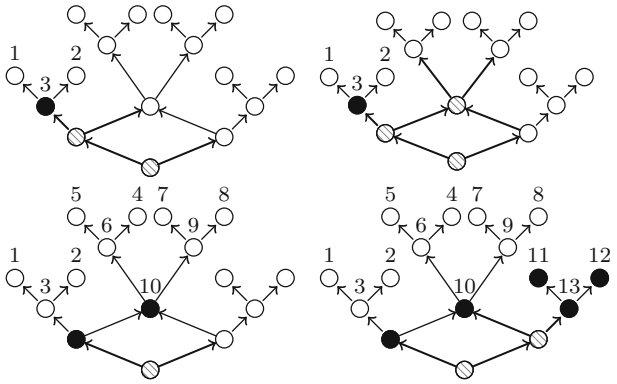

Fig. 4 Bottom-up pebbling

no point will more than five pebbles be used for pebbling the root node, which is shown in the bottom right picture of the figure. This is independent of the heuristic choices.

Note that bottom-up pebbling creates a memory overhead for storing the premise decisions in nodes until all alternatives are expanded. The number of decisions to be stored is limited by the depth of the proof. Furthermore, it is not necessary to store all information contained in a proof node. The algorithm only needs to remember which premises were already processed. The aim of our work is the post-processing of proofs and construction of proof node orders. We are interested in the memory consumption when using such an order for proof processing. The memory requirement of producing it is not our main focus. Therefore, we ignore the memory overhead in the further discussion.

5.3 Complexity

The presented algorithms have linear time and space complexity in the size of the proof n.

Both bottom-up and top-down pebbling visit every node exactly once, which results in n recursive function calls for bottom-up and n iterations through the main while loop for top-down. The inner loop of top-down pebbling in line 7 is executed at most $2n$ times, since every node is child of at most two parent nodes. For every recursive call, the inner loop of bottom-up pebbling in line 4 is executed only a constant number of times, since every N is a subset of the parent nodes, which is at most of size 2.

The space complexity of our algorithms is linear, since in the worst case all nodes have to be kept in memory during proof processing. However, as our experiments show, typically the space requirements of proofs are much lower than their length.

We assume that heuristic decisions use constant time and space, which is the case for the presented heuristics in Sect. 6.

Algorithm 2: `Bottom-Up Pebbling`

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

1 $D = D \cup \{v\}$;
2 $N = P_v^{\varphi} \setminus D$; `// Visit only unprocessed premises`
3 $S_1 = S$;
4 **while** N *is not empty* **do**
5 choose $p \in N$ heuristically;
6 $N = N \setminus p$;
7 $S_1 = S_1$ :::`Bottom-Up Pebbling`(φ, p, D, S);
8 **return** S_1 ::: (v);

5.4 Remarks on top-down and bottom-up pebbling

The experiments presented in Sect. 7 show that in practice, bottom-up pebbling performs much better than top-down. In the following, we present two principles that result in pebbling strategies with small pebbling numbers and are likely to be violated by the top-down pebbling algorithm.

Firstly, a pebbling strategy 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 design, 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 child nodes; therefore, it is hard to make local choices. The algorithm simply does not know which pebbleable nodes are close to other pebbled ones. Example 4 illustrates this principle.

Secondly, pebbling strategies should process 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 shared between subproofs. Example 6 illustrates this principle.

Example 6 Figure 5 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 $\prec_{1-j}$, then by $\prec_j$ followed by

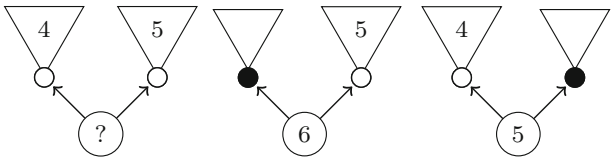

Fig. 5 Spacious subproof first

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.

6 Heuristics

The presented algorithms are parametrized by a heuristic, selecting one node v out of a set of nodes 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 12 (*Heuristic*) 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 $e_h : V \to S_h$. The *choice* of the heuristic for a set $N \subseteq V$ is some $v \in N$ such that $v = \text{argmax}_{v \in N} e_h(v)$

The argmax of $e_h(v)$ is not unique in general. In practice, we simply use another heuristic to decide ties and eventually have to decide upon some trivial criteria as, for example, address in memory. We do not elaborate on the results of using different heuristics to decide ties.

In the following paragraphs, we present and motivate heuristics that rank nodes based on structural characteristics of proofs.

6.1 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 6 shows the idea behind pebbling spacious subproofs early.

6.2 LastChild heuristic ("*Lc*")

As discussed in Sect. 3 in the proof of Theorem 1, 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 *LastChild* 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 for which a node is the last child, the proof has to be traversed once, before constructing the new order, 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, often the proof structure ensures that a node is the last child of another node irrespective of the topological order.

6.3 Node Distance heuristic ("Dist(r)")

In Example 4 and Sect. 5.4, 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. The sphere $K_r^G(v)$ with radius r around the node v in the graph $G = (V, E)$ is defined as the set of nodes in V that is connected to v via at most r undirected edges. The heuristic uses the following functions based on the spheres:

$$d(v) := \begin{cases} -D & \text{where } D = \min\{r \mid K_r^G(v) \text{ contains a pebbled node}\} \\ -\infty & \text{if no such } D \text{ exists} \end{cases}$$

$$s(v) := |K_{-d(v)}^G(v)|$$

$$l(v) := \max_{\prec} K_{-d(v)}^G(v)$$

$$e_h(v) := (d(v), s(v), l(v))$$

where $\prec$ denotes the order of previously pebbled nodes. So $S_h = \mathbb{Z} \cup \{\text{infty}\} \times \mathbb{N} \times V$ together with the lexicographic order using, respectively, the natural smaller relation $<$ on $\mathbb{N}$ and $\mathbb{Z}$, where ∞ is an element that is bigger than all others, and $\prec$ on V. The spheres $K_r(v)$ can grow exponentially in r. Therefore, the maximum radius has to be kept small.

6.4 Decay heuristics ("*Dc*(h_u, γ, d, c)")

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

Table 1 Proof benchmark sets, where length is measured in number of nodes

Name	No. of proofs	Max length	Avg length
TraceCheck_1	2239	90,756	5423
TraceCheck_2	215	1,768,249	268,863
SMT_1	4187	2,241,042	103,162
SMT_2	914	120,075	5391

decay factor $\gamma \in \mathbb{R}^+ \cup \{0\}$, a recursion depth $d \in \mathbb{N}$ and a combining function $c : S_u^n \to S_u$ for $n \in \mathbb{N}$. The resulting heuristic node evaluation function e_h is defined recursively, using function r:

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

7 Experiments

The experiments on the space compression algorithm were performed on four disjoint sets of proof benchmarks (Table 1). TraceCheck_1 and TraceCheck_2 contain proofs produced by the SAT-solver PicoSAT [8] on unsatisfiable benchmarks from SATLIB. The proofs are in the TraceCheck proof format, which is one of the three formats accepted at the *Certified UNSAT* track of the SAT-competition. SMT_1 and SMT_2 contain proofs produced by the SMT-solver VeriT [14] on unsatisfiable problems from the QF_UF (quantifier-free with uninterpreted function symbols) division of the SMT-LIB. The smaller sets TraceCheck_2 and SMT_2 contain proofs of a first set of experiments that tested all heuristics and parameters. To this end, we split the alphabetically ordered list of all available proofs into 200 equally sized lists and processed them in parallel until our global timeout was reached. TraceCheck_1 and SMT_1 contain all remaining proofs. The first and second set of experiments used a global time limit of 100 and 30 h, respectively. We did not impose any timeout for processing each single proof, since all proofs could be processed in a reasonable amount of time. These proofs are in a proof format that resembles SMT-LIB's problem format. Besides axiom nodes containing input clauses, they also have nullary (i.e., premise-less) equality inferences containing instances of the reflexivity, symmetry, transitivity and congruence axioms of equality. For the purposes of the algorithms evaluated here, nullary equality inferences can be treated in exactly the same way as an ordinary axiom and, therefore, the proofs can be considered pure resolution proofs.

Table 2 Experimental results, where **RP** denotes the relative performance according to Formula 1

Algorithm heuristic	RP (%)	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 2 summarizes the main 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 SMT_2 only.[2] The relative performance is calculated according to Formula 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. Intuitively, the relative performance describes how well a given heuristic performed in comparison with the mean performance of all heuristics. The time used to construct orders is measured in processed nodes per millisecond. Both columns show the best and worst result in boldface.

$$\text{rp}(f, P, G) = \frac{1}{|P|} * \sum_{\varphi \in P} \left(1 - \frac{s(\varphi, f(\varphi))}{\text{mean}_{g \in G} s(\varphi, g(\varphi))}\right) \quad (1)$$

Table 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 Fig. 6, 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 algorithms. The results for top-down range far beyond 15,000, but to display the discrepancy between the two algorithms, the plot scales from 0 to 15,000 on both axes. The largest 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

[2] The performance observed on these benchmark sets was not promising enough to justify spending more computational resources of the Vienna Scientific Cluster to evaluate these heuristics on the larger benchmark sets TraceCheck_1 and SMT_1.

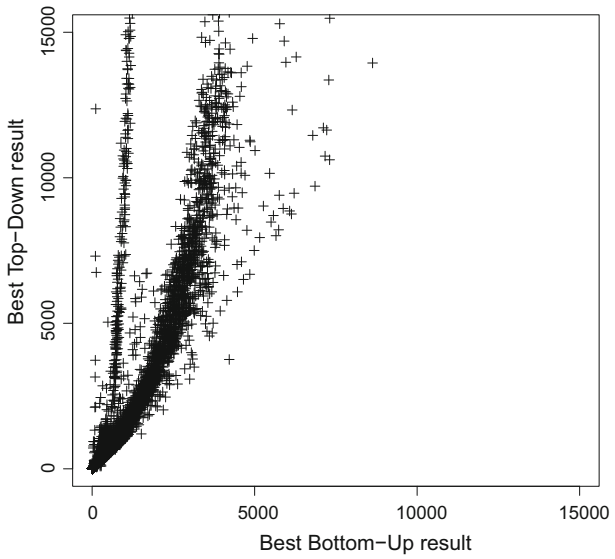

Fig. 6 Space measures of best bottom-up and top-down result

Table 3 Improvement of last child using Decay heuristic, where **PI** denotes the performance improvement against no Decay heuristic

Decay (γ)	Depth (d)	Combin. (c)	PI(%)	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

the worst results, which could be due to the fact that the radius is too small for large proofs with thousands of nodes.

Table 3 summarizes results of the Decay heuristic with the best results highlighted in boldface. Decay heuristics were tested with the bottom-up algorithm, using last child 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 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 high.

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 2. Most likely, the reason is the number of comparisons made by the algorithms. For bottom-up the set N of possible choices consists of the premises of a single node only, i.e., $|N| \in \{0, 2\}$. 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.

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 during top-down proof processing (e.g., top-down proof checking).

8 Conclusion

The problem of compressing proofs in space has been reduced to finding strategies in a pebbling game, for which finding the optimal strategy is known to be NP-complete. Therefore, two heuristic algorithms 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 algorithms are parameterized by a heuristic function for selecting nodes. The best performances are achieved with the simplest heuristics (i.e., LastChild and Number of Children). More sophisticated heuristics provided little extra compression but had a high cost in execution time.

When a proof is compressed in space and deletion information is added, top-down proof checkers (and proof processors in general) know when they do not need to keep a node in memory anymore. In the benchmarks considered here, a top-down proof checker would need on average 44.1 times less memory for storing nodes when given a proof with deletion information added than if they had to keep all nodes in memory.

One limitation of the presented experiments is that they compare only the space of proofs compressed by the proposed heuristics, but do not compare the space of the input proof using a topological order implied by the proof file generated by the solver. SMT-proof files generated by VeriT do indeed imply a clear and rigid topological order (a clause appearing in line $n+1$ depends only on clauses in lines 1 to n, and the premises used to derive the clause are supposed to be resolved in a left-associative manner). However, SAT proofs in the TraceCheck format are more liberal (resolution chains do not need to be left-associative (and usually are not) and a clause in line n may depend on a clause in a

line $m > n$) and, therefore, they do not have a single clear topological order. Another reason for not considering any implied topological order is technical; Skeptik's underlying data structures were designed to accept only binary resolutions. When a proof file is parsed, n-ary resolution chains are converted into sequences of binary resolutions and any topological order implied by the chains is lost. Furthermore, even if Skeptik did keep information about the chains, the chains would eventually be broken when other proof compression algorithms (which rely on a binary resolution structure) were applied to the proof. Therefore, the limitation is irrelevant in this scenario.

Future work could investigate space compression heuristics that take advantage of the particular shape of resolution chains generated by conflict graph analysis, thereby addressing the limitation above. Such future work would be particularly relevant if the space compression algorithms were implemented directly into a SAT- or SMT-solver to provide proofs with small space right away.

Acknowledgements Open access funding provided by Austrian Science Fund (FWF). Andreas Fellner was supported by the Google Summer of Code 2013 program, by FFG Project No. 845582 (TRUCONF) and was co-funded by FWF Project W1255-N23. Bruno Woltzenlogel Paleo was supported by the Austrian Science Fund, Project P24300.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: The matita interactive theorem prover. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction—CADE-23—23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31–August 5, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6803, pp. 64–69. Springer (2011)
2. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory. http://www.lsv.fr/~dowek/Publi/expressing.pdf (2016)
3. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-time reductions of resolution proofs. In: Haifa Verification Conference, pp. 114–128 (2008)
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification—23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14–20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011)
5. Barrett, C., Fontaine, P., de Moura, L.: Proofs in satisfiability modulo theories. In: Woltzenlogel Paleo, B., Delahaye, D. (eds.) All About Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55. College Publications, London, UK. http://www.collegepublications.co.uk/logic/mlf/?00023 (2015)
6. Ben-Sasson, E.: Size space tradeoffs for resolution. In: STOC, pp. 457–464 (2002)
7. Ben-Sasson, E., Nordström, J.: Short proofs may be spacious: an optimal separation of space and length in resolution. Electron. Colloq. Comput. Complex. **16**, 2 (2009)
8. Biere, A.: Picosat essentials. JSAT **4**(2–4), 75–97 (2008)
9. Biere, A., Heule, M.: Satisfiability solvers. In: Woltzenlogel Paleo, B., Delahaye, D. (eds.) All About Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55. College Publications, London, UK. http://www.collegepublications.co.uk/logic/mlf/?00023 (2015)
10. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
11. Blanchette, J.C., Urban, J. (eds.): Third International Workshop on Proof Exchange for Theorem Proving, PxTP 2013, Lake Placid, NY, USA, June 9–10, 2013, EPiC Series in Computing, vol. 14. EasyChair. http://www.easychair.org/publications/?page=771427785 (2013)
12. Boudou, J., Fellner, A., Woltzenlogel Paleo, B.: Skeptik: A proof compression system. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR. Lecture Notes in Computer Science, vol. 8562, pp. 374–380. Springer. doi:10.1007/978-3-319-08587-6 (2014)
13. Boudou, J., Woltzenlogel Paleo, B.: Compression of propositional resolution proofs by lowering subproofs. In: Galmiche and Larchey-Wendling [30], pp. 59–73. doi:10.1007/978-3-642-40537-2_7 (2013)
14. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: Verit: an open, trustable and efficient SMT-solver. In: CADE, pp. 151–156 (2009)
15. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, pp. 1–5. ACM (2009)
16. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Theory and Applications of Satisfiability Testing—SAT 2010, pp. 44–57. Springer (2010)
17. Burel, G.: A shallow embedding of resolution and superposition proofs into the $\lambda\Pi$-calculus modulo. In: Blanchette, J.C., Urban, J. (eds.) Third International Workshop on Proof Exchange for Theorem Proving, PxTP 2013, Lake Placid, NY, USA, June 9–10, 2013. EPiC Series in Computing, vol. 14, pp. 43–57. EasyChair. http://www.easychair.org/publications/?page=1045498133 (2013)
18. Chan, S.M.: Pebble games and complexity. Technical Report UCB/EECS-2013-145, University of California, Berkeley, EECS Department, University of California, Berkeley, USA (2013)
19. Chihani, Z., Libal, T., Reis, G.: The proof certifier checkers. In: de Nivelle, H. (ed.) Automated Reasoning with Analytic Tableaux and Related Methods—24th International Conference, TABLEAUX 2015, Wrocław, Poland, September 21–24, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9323, pp. 201–210. Springer. doi:10.1007/978-3-319-24312-2_14 (2015)
20. Chihani, Z., Miller, D., Renaud, F.: Foundational proof certificates in first-order logic. In: Bonacina, M.P. (ed.) Automated Deduction—CADE-24—24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9–14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7898, pp. 162–177. Springer (2013)
21. Cotton, S.: Two techniques for minimizing resolution proofs. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 306–312. Springer (2010)
22. Dowek, G., Dubois, C., Pientka, B., Rabe, F. Universality of Proofs (Dagstuhl Seminar 16421). Dagstuhl Rep. **6**(10), 75–98 (2017) doi:10.4230/DagRep.6.10.75

23. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: Barthe, G., Hermenegildo, M.V. (eds.) Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17–19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 5944, pp. 129–145. Springer (2010)
24. Dunchev, C., Leitsch, A., Libal, T., Riener, M., Rukhaia, M., Weller, D., Paleo, B.W.: PROOFTOOL: a GUI for the GAPT framework. In: Kaliszyk, C., Lüth, C. (eds.) Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012. EPTCS, vol. 118, pp. 1–14. doi:10.4204/EPTCS.118.1 (2012)
25. Dunchev, T., Leitsch, A., Libal, T., Weller, D., Woltzenlogel Paleo, B.: System description: the proof transformation system CERES. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16–19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6173, pp. 427–433. Springer. doi:10.1007/978-3-642-14203-1_36 (2010)
26. Ebner, G., Hetzl, S., Reis, G., Riener, M., Wolfsteiner, S., Zivota, S.: System description: GAPT 2.0. In: Olivetti, N., Tiwari, A. (eds.) Automated Reasoning—8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9706, pp. 293–301. Springer. doi:10.1007/978-3-319-40229-1_20 (2016)
27. van Emde Boas, P., van Leeuwen, J.: Move rules and trade-offs in the pebble game. In: Weihrauch, K. (ed.) Theoretical Computer Science. Lecture Notes in Computer Science, vol. 67, pp. 101–112. Springer, Berlin (1979)
28. Esteban, J.L., Torán, J.: Space bounds for resolution. Inf. Comput. **171**(1), 84–97 (2001)
29. Fontaine, P., Merz, S., Woltzenlogel Paleo, B.: Compression of propositional resolution proofs via partial regularization. In: CADE, pp. 237–251 (2011)
30. Galmiche, D., Larchey-Wendling, D. (eds.): Automated Reasoning with Analytic Tableaux and Related Methods—22nd International Conference, TABLEAUX 2013, Nancy, France, September 16–19, 2013. Proceedings, Lecture Notes in Computer Science, vol. 8123. Springer. doi:10.1007/978-3-642-40537-2 (2013)
31. Gilbert, J.R., Lengauer, T., Tarjan, R.E.: The pebbling problem is complete in polynomial space. SIAM J. Comput. **9**(3), 513–524 (1980)
32. Gorzny, J., Woltzenlogel Paleo, B.: Towards the compression of first-order resolution proofs by lowering unit clauses. In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction—CADE-25—25th International Conference on Automated Deduction, Berlin, Germany, August 1–7, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9195, pp. 356–366. Springer. doi:10.1007/978-3-319-21401-6_24 (2015)
33. Hertel, P., Pitassi, T.: Black-white pebbling is PSPACE-complete. Electron. Colloq. Comput. Complex. **14**(044). http://eccc.hpiweb.de/eccc-reports/2007/TR07-044/index.html (2007)
34. Hetzl, S., Leitsch, A., Reis, G., Weller, D.: Algorithmic introduction of quantified cuts. Theor. Comput. Sci. **549**, 1–16 (2014). doi:10.1016/j.tcs.2014.05.018
35. Hetzl, S., Leitsch, A., Weller, D., Woltzenlogel Paleo, B.: Herbrand sequent extraction. In: Intelligent Computer Mathematics, 9th International Conference, AISC 2008, 15th Symposium, Calculemus 2008, 7th International Conference, MKM 2008, Birmingham, UK, July 28–August 1, 2008. Proceedings, pp. 462–477. doi:10.1007/978-3-540-85110-3_38 (2008)
36. Hetzl, S., Libal, T., Riener, M., Rukhaia, M.: Understanding resolution proofs through Herbrand's theorem. In: Galmiche and Larchey-Wendling [30], pp. 157–171. doi:10.1007/978-3-642-40537-2_15 (2013)
37. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR abs/1610.06229. http://arxiv.org/abs/1610.06229 (2016)
38. Hofferek, G., Gupta, A., Könighofer, B., Jiang, J.H.R., Bloem, R.: Synthesizing multiple Boolean functions using interpolation on a single proof. CoRR abs/1308.4767 (2013)
39. Huet, G., Paulin-Mohring, C., et al.: The Coq proof assistant reference manual. Part Coq Syst. Version **6**(1). https://pdfs.semanticscholar.org/fa95/827bfe2b83c2e27e4d17214fb22eca13cc87.pdf (2000)
40. Kaliszyk, C., Paskevich, A. (eds.): Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2–3, 2015, EPTCS, vol. 186. doi:10.4204/EPTCS.186 (2015)
41. Kaliszyk, C., Urban, J.: HOL(y)Hammer: online ATP service for HOL Light. Math. Comput. Sci. **9**(1), 5–22 (2015)
42. Kasai, T., Adachi, A., Iwata, S.: Classes of pebble games and complete problems. SIAM J. Comput. **8**(4), 574–586 (1979)
43. Leitsch, A.: The Resolution Calculus. Texts in Theoretical Computer Science. Springer, Berlin (1997)
44. Libal, T., Riener, M., Rukhaia, M.: Advanced proof viewing in proof tool. In: Benzmüller, C., Paleo, B.W. (eds.) Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014. EPTCS, vol. 167, pp. 35–47. doi:10.4204/EPTCS.167.6 (2014)
45. Miller, D.: Foundational proof certificates. In: Woltzenlogel Paleo, B., Delahaye, D. (eds.) All About Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55. College Publications, London, UK. http://www.collegepublications.co.uk/logic/mlf/?00023 (2015)
46. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/programming-higher-order-logic?format=HB (2012)
47. Necula, G.C.: Proof-carrying code. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15–17 January 1997. pp. 106–119. ACM Press (1997)
48. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)
49. Nordström, J.: Narrow proofs may be spacious: separating space and width in resolution. SIAM J. Comput. **39**(1), 59–121 (2009)
50. Paulson, L.C., Blanchette, J.C.: Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. IWIL-2010 (2010)
51. Philipp, T., Rebola-Pardo, A.: Towards a semantics of unsatisfiability proofs with inprocessing (2017)
52. Pientka, B.: Beluga: Programming with dependent types, contextual data, and contexts. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19–21, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6009, pp. 1–12. Springer (2010)
53. Pippenger, N.: Comparative schematology and pebbling with auxiliary pushdowns (preliminary version). In: Miller, R.E., Ginsburg, S., Burkhard, W.A., Lipton, R.J. (eds.) STOC. pp. 351–356. ACM (1980)
54. Pippenger, N.: Advances in Pebbling. Springer, Berlin (1982)
55. Reis, G.: Importing SMT and connection proofs as expansion trees. In: Kaliszyk, C., Paskevich, A. (eds.) Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2–3, 2015. EPTCS, vol. 186, pp. 3–10. doi:10.4204/EPTCS.186.3 (2015)
56. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)

57. Rollini, S., Bruttomesso, R., Sharygina, N.: An efficient and flexible approach to resolution proof reduction. In: Hardware and Software: Verification and Testing—6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4–7, 2010. Revised Selected Papers, pp. 182–196. doi:10.1007/978-3-642-19583-9_17 (2010)
58. Schürmann, C.: The twelf proof assistant. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17–20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 79–83. Springer (2009)
59. Sethi, R.: Complete register allocation problems. SIAM J. Comput. **4**(3), 226–248 (1975)
60. Slaney, J., Paleo, B.W.: Conflict resolution: a first-order resolution calculus with decision literals and conflict-driven clause learning. CoRR abs/1602.04568. http://arxiv.org/abs/1602.04568 (2016)
61. Walker, S.A., Strong, H.R.: Characterizations of flowchartable recursions. J. Comput. Syst. Sci. **7**(4), 404–447 (1973)
62. Woltzenlogel Paleo, B.: Herbrand Sequent Extraction [M.Sc. Thesis]. VDM-Verlag, Saarbrucken, Germany (2008)
63. Woltzenlogel Paleo, B.: Atomic cut introduction by resolution: proof structuring and compression. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning—16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 463–480. Springer. doi:10.1007/978-3-642-17511-4_26 (2010)
64. Woltzenlogel Paleo, B.: Contextual natural deduction. In: Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6–8, 2013. Proceedings. pp. 372–386. doi:10.1007/978-3-642-35722-0_27 (2013)
65. Woltzenlogel Paleo, B.: Implementation and evaluation of contextual natural deduction for minimal logic. In: Perspectives of System Informatics—10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24–27, 2015, Revised Selected Papers, pp. 314–324. doi:10.1007/978-3-319-41579-6_24 (2015)
66. Woltzenlogel Paleo, B., Delahaye, D.: All About Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55. College Publications, London, UK. http://www.collegepublications.co.uk/logic/mlf/?00023 (2015)