International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Greedy Pebbling for Proof Space Compression

Andreas Fellner · Bruno Woltzenlogel Paleo

the date of receipt and acceptance should be inserted later

Abstract This paper describes algorithms and heuristics for playing a *Pebbling Game*. Playing the game with a small number of pebbles is analogous to processing a proof with a small amount of available memory. Here this analogy is exploited: new pebbling algorithms are conceived and evaluated on the task of compressing the space of thousands of propositional resolution proofs generated by SAT- and SMT-solvers. We would like to thank Armin Biere for clarifying why resolution chains are not left-associative in the TraceCheck proof format.

Keywords Proof Compression, Memory Consumption, Resolution, Pebbling Games

1 Acknowledgments

Andreas Fellner was supported by the Google Summer of Code 2013 program. Bruno Woltzenlogel Paleo was supported by the Austrian Science Fund, project P24300.

2 Introduction

SAT- and SMT-solvers are among the most efficient tools for the verification and synthesis of software [6, 14]. However, proofs generated by SAT- and SMT-solvers can be huge. Checking their correctness or extracting information (e.g. unsatisfiable cores, interpolants) from them can not only take a long time but also consume a lot of memory. In an ongoing project for interpolant-based controller synthesis [14], for example, extracting an interpolant from an SMT-proof took hours and reached the limit of memory (256GB) available in a single node of the computer cluster used in the project. This issue is also relevant in application scenarios in which the proof consumer, who is interested in independently processing proofs, might have less available memory than the proof producer.

Typically, proof formats 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. To address this issue, recently proposed proof formats for SAT-proofs such as DRAT and BDRUP enrich the older RUP proof format with node deletion instructions [23]. Other proof formats, such as the TraceCheck format [5] or formats for SMT-proofs [1], could also be enriched analogously. When generating a proof file eagerly (i.e. writing learned clauses immediately to the proof file) a SAT- or SMT-solver can add a deletion instruction for every clause that is deleted by the periodic clean-up of its database of derived learned clauses.

This paper explores the possibility of post-processing a proof in order to increase the amount of deletion instructions in the proof file. 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* [15,12]. In Section 3 we define propositional resolution proofs and make the notion of processing a proof formal.

Andreas Fellner
E-mail: afellner@forsyte.tuwien.ac.at
Bruno Woltzenlogel Paleo
E-mail: bruno@logic.at

Theory and Logic Group
Institute for Computer Languages
Vienna University of Technology

The particular version of pebbling game relevant for proof processing is defined precisely in Section 4, where we also explain the analogy to proof processing in detail and define the space measure of proofs. The proposed pebbling algorithms are greedy (Section 6) and based on heuristics (Section 7). As discussed in Sections 4 and 5, 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) [24]. 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 (Section 8) evaluate the proposed algorithms on thousands of SAT- and SMT-proofs.

3 Propositional Resolution Proofs

Resolution is among the most prominent formal calculi for automated deduction and goes back to Robinson [20]. 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 [6]. For an extensive discussion of propositional and first-order logic resolution, we refer the reader to [16].

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 $C_1 \setminus \{\ell\} \cup C_2 \setminus \{\overline{\ell}\}$ 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 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 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 $x \in V_R : \mathcal{L}(x) = \mathcal{L}_2(x)$, $V = (V_L \cup V_R) \cup \{v\}$, $E = E_L \cup E_R \cup \{(v_L, v), (v_R, v)\}$.

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 [7,6]. The reuse of proof nodes plays a central role in proof compression [11].

Several measures can be defined on proofs. The relevant measure for this work is space, which is defined in Section 4. 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 unsatisfiable 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 $\{\neg x_1\}$. The resulting proof is displayed in Figure 1.

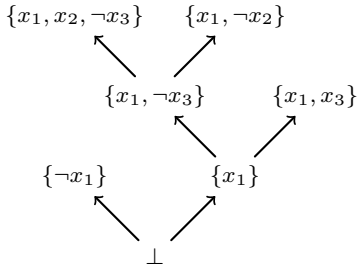

Fig. 1: Proof of Φ's unsatisfiability

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

$$\alpha(v, f) = \begin{cases} f(v, \alpha(pr_1, f), \alpha(pr_2, f)) & \text{if } P_v^\varphi = \{pr_1, pr_2\} \\ g_f(v) & \text{otherwise} \end{cases}$$

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

Example 2 Checking the correctness of a 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.

$$f(v, w_1, w_2) = \begin{cases} \top & \text{if } v \text{ has no premises} \\ w_1 \wedge w_2 & \text{if } P_v^\varphi = \{pr_1, pr_2\} \text{ and} \\ & \quad \mathcal{L}(v) \text{ is the resolvent of} \\ & \quad \mathcal{L}(pr_1) \text{ and } \mathcal{L}(pr_2) \\ \bot & \text{otherwise} \end{cases}$$

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

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.

4 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 1970's to model programming language expressiveness [18,22] and compiler construction [21]. More recently, pebbling games have been used to investigate various questions in parallel complexity [8] and proof complexity [3,10,17]. They are used to obtain bounds for space- and time-requirements and trade-offs between the two measures [9,2].

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 pebbles.

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 Section 6, 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 Boundend 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

$$Peb_i^\sigma := \{v \in V \mid \exists j \leq i : \sigma_j = p(v) \wedge \forall k : j < k \leq i : \sigma_k \neq u(v)\}$$

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

$$UPeb_i^\sigma := \{v \in V \mid \nexists j \leq i : \sigma_j = u(v) \wedge \forall c \in C_v^\varphi c \in Peb_i^\sigma\}$$

The following definition allows to measure how many pebbles are required 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. $max_{i \in \{1 \ldots n\}} |Peb_i^\sigma|$. 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 [13, 19] in two aspects. Firstly, the Black Pebbling Game does not include rule 3. Excluding this rule allows for pebbling strategies with lower pebbling numbers ([21] has an example on page 1), at the expense of an exponential upper bound on the number of rounds [9]. 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 [9]. 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 [12] and NP-complete when rule 3 is included [21].

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 premises lower than their children corresponds to rule 1 of the Bounded Pebbling Game. The antisymmetry together with the fact that $V = \{v_1, \ldots, v_n\}$ correspond 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 Topological Pebbling Strategy) The *canonical topological 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 recur-

sively:

$$\sigma_1 = p(v_1)$$
$$\sigma_i = \begin{cases} u(v) & \text{if } UPeb_i^\sigma \neq \emptyset, \text{ where} \\ & \quad v = min_\prec(UPeb_i^\sigma) \\ p(v) & \text{otherwise, where} \\ & \quad v = min_\prec(w \mid \text{ for all } l < i : \sigma_l \neq p(w)) \end{cases}$$

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 topological 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 comsumer 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.

Example 3 Consider the proof displayed in Figure 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.

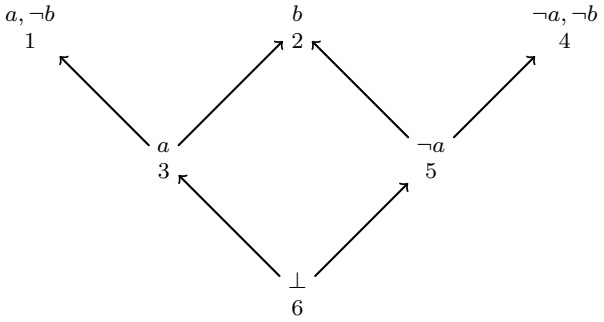

Fig. 2: A Simple Proof

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 $l(\varphi_m) \in O(m)$ and $|T(\varphi_m)| \in \Omega(m!)$, where $T(\varphi_m)$ is the set of possible topological orders for φ_m.*

Proof Let φ_m be a perfect binary tree with m axioms. Clearly, $l(\varphi_m) = 2m - 1$. Let $(v_1, \ldots, v_n)$ be a topological order for φ_m. Let $A_{\varphi_m} = \{v_{k_1}, \ldots, v_{k_m}\}$, then $(v_{k_1}, \ldots, v_{k_m}, v_{l_1}, \ldots, v_{l_{n-m}})$, where the indexes are such that $(l_1, \ldots, l_{n-m}) = (1, \ldots, n) \setminus (k_1, \ldots, k_m)$, is a topological order as well.

Likewise, $(v_{\pi(k_1)}, \ldots, v_{\pi(k_m)}, v_{l_1}, \ldots, v_{l_{n-m}})$ is a topological order, for every permutation π of $\{k_1, \ldots, k_m\}$. There are $m!$ such permutations, so the overall number of topological orders is at least factorial in m (and also in n).

□

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 $(\varphi_2, \varphi_4, \ldots)$ 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 m axioms, where $m > 1$. Let $\delta = (v_{k_1}, \ldots, v_{k_m}, v_{l_1}, \ldots, v_{l_{n-m}})$ be the topological order, where indices are defined as in the proof of Theorem 3. The strategy initially pebbles all axioms, making no node available for unpebbling. Only after pebbling one additional node, two axioms can be unpebbled. Therefore, we have that $n_\delta = m + 1$.

Let γ be the strategy that processes φ_m from left to right. We show by induction on $m : n_\gamma = \log_2(m) + 2$. The base case is $m = 2$. The strategy γ has to pebble both axioms first, before being able to pebble the root node. Therefore, we have $n_\gamma = 3 = \log_2(2) + 2$ for φ_2.

Let φ_m be a perfect tree with $m = m' * 2$ axioms, which has a left and a right subproof, which are perfect bianry trees with m' axioms. By induction hypothesis, we have that γ needs $\log_2(m') + 2$ pebbles on the left subproof. One pebble remains on the root of the left subproof, while processing the right subproof. Therefore, we have $n_\gamma = \log_2(m') + 3 = \log_2(\frac{m}{2}) + 3 = \log_2(m) + 2$ for φ_m.

We have $n_\delta = m + 1 = O(2^{\log_2(m)+2}) = O(2^{n_\gamma})$.

□

5 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 [6]. 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) 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 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 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. The following theorem states the correctness of the 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 or equal k

6 Greedy Pebbling Algorithms

Theorem 3 and the remarks in the end of Section 5 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 Section 7, but differ in the traversal direction in which the algorithms operate on proofs.

6.1 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.

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 Figure 3 and suppose that Top-Down Pebbling has already pebbled

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;
```

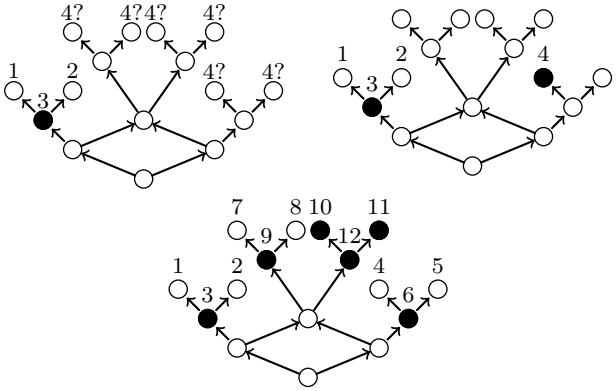

Fig. 3: Top-Down Pebbling

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 3. However the pebbling number of this proof is five.

6.2 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 `BUpebble`$(\varphi, v, \emptyset, ())$. 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 Section 3 with the addition of a visit order of the premises. Also previously visited nodes are not visited again.

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 ∪ {v};
2 N = P_v^φ \ D;   // Visit only unprocessed premises
3 S_1 = S;
4 while N is not empty do
5     choose p ∈ N heuristically;
6     N = N \ p;
7     S_1 = S_1 ::: BottomUpPebbling(φ, p, D, S);
8 return S_1 ::: (v);
```

Example 5 Figure 4 shows part of an execution of Bottom-Up Pebbling on the same proof as presented in Figure 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 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 independent of the heuristic choices.

6.3 Remarks about Top-Down and Bottom-Up Pebbling

The experiments presented in Section 8 show that in practice, Bottom-Up Pebbling performs much better than Top-Down. Example 4 shows two principles that

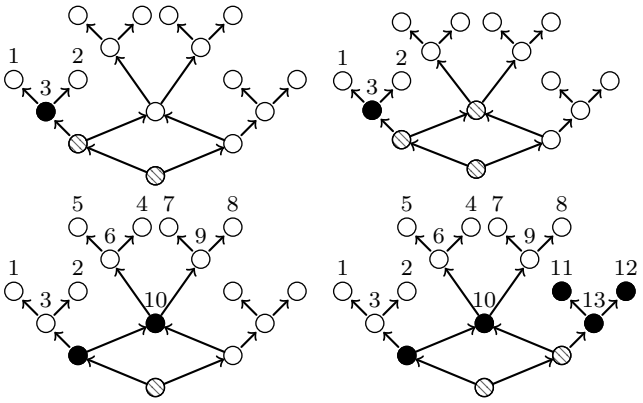

Fig. 4: Bottom-Up Pebbling

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.

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 shared between subproofs. The principle is demonstrated in the following example.

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 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.

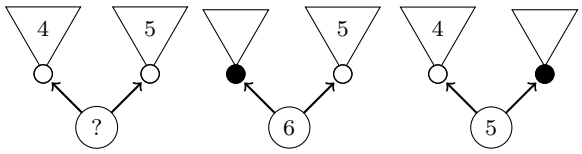

Fig. 5: Spacious subproof first

7 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 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 $v = argmax_{v \in N} e_h(v)$

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 upon structural characteristics proofs.

7.1 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.

7.2 Last Child Heuristic ("Lc")

As discussed in Section 4 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 *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, 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.

7.3 Node Distance Heuristic ("$Dist(r)$")

In Example 4 and Section 6.3 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 $S_h = \mathbb{Z} \cup \{\infty\} \times \mathbb{N} \times P$ 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

Table 1: Proof Benchmark Sets, where length is measured in number of nodes

Name	**#Proofs**	**Max length**	**Avg length**
TraceCheck_1	2239	90756	5423
TraceCheck_2	215	1768249	268863
SMT_1	4187	2241042	103162
SMT_2	914	120075	5391

all others, and $\prec$ on N. The spheres $K_r(v)$ can grow exponentially in r. Therefore the maximum radius has to be kept small.

7.4 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 recursively, using function r:

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

8 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 [4] 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 [7] on unsatisfiable problems from the SMT-LIB. These proofs 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 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. 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. The time used to construct orders is measured in processed nodes per millisecond. Both columns show the best and worst result in boldface.

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 3: Improvement of LastChild using Decay Heuristic, where **PI** denotes the performance improvement against no decay heuristic

Decay γ	**Depth** d	**Combin.** *com*	**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

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

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 Figure 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 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 largest best space measure for Top-Down is 131 451, whereas

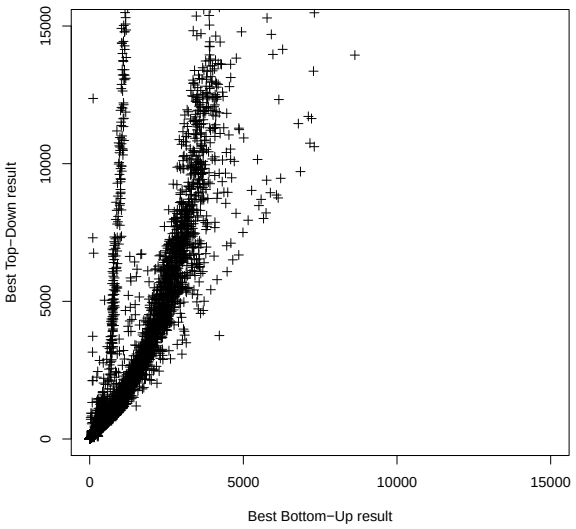

Fig. 6: Space measures of best Bottom-Up and Top-Down result

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 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 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 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.

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 during proof processing.

9 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 for compressing proofs in 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 algorithms are parametrized 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. Furthermore, the methods could be implemented directly into a SAT- or SMT-solver to provide proofs with small space right away.

ToDo: fix overfull hboxes

References

1. 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. College Publications (to appear in 2015)
2. Ben-Sasson, E.: Size space tradeoffs for resolution. In: STOC. pp. 457–464 (2002)
3. Ben-Sasson, E., Nordström, J.: Short proofs may be spacious: An optimal separation of space and length in resolution. Electronic Colloquium on Computational Complexity (ECCC) 16, 2 (2009)
4. Biere, A.: Picosat essentials. JSAT 4(2-4), 75–97 (2008)
5. Biere, A., Heule, M.: Satisfiability solvers. In: Woltzenlogel Paleo, B., Delahaye, D. (eds.) All about Proofs, Proofs for All. College Publications (to appear in 2015)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)

7. 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)
8. Chan, S.M.: Pebble games and complexity (2013)
9. van Emde Boas, P., van Leeuwen, J.: Move rules and trade-offs in the pebble game. In: Theoretical Computer Science. pp. 101–112 (1979)
10. Esteban, J.L., Torán, J.: Space bounds for resolution. Inf. Comput. 171(1), 84–97 (2001)
11. Fontaine, P., Merz, S., Paleo, B.W.: Compression of propositional resolution proofs via partial regularization. In: CADE. pp. 237–251 (2011)
12. Gilbert, J.R., Lengauer, T., Tarjan, R.E.: The pebbling problem is complete in polynomial space. SIAM J. Comput. 9(3), 513–524 (1980)
13. Hertel, P., Pitassi, T.: Black-white pebbling is pspace-complete. Electronic Colloquium on Computational Complexity (ECCC) 14(044) (2007)
14. 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)
15. Kasai, T., Adachi, A., Iwata, S.: Classes of pebble games and complete problems. SIAM J. Comput. 8(4), 574–586 (1979)
16. Leitsch, A.: The resolution calculus. Texts in theoretical computer science, Springer (1997)
17. Nordström, J.: Narrow proofs may be spacious: Separating space and width in resolution. SIAM J. Comput. 39(1), 59–121 (2009)
18. Pippenger, N.: Comparative schematology and pebbling with auxiliary pushdowns (preliminary version). In: STOC. pp. 351–356 (1980)
19. Pippenger, N.: Advances in pebbling. Springer (1982)
20. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM 12(1), 23–41 (1965)
21. Sethi, R.: Complete register allocation problems. SIAM J. Comput. 4(3), 226–248 (1975)
22. Walker, S.A., Strong, H.R.: Characterizations of flowchartable recursions. J. Comput. Syst. Sci. 7(4), 404–447 (1973)
23. Wetzler, N., Heule, M., Hunt, WarrenA., J.: Drat-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing SAT 2014, Lecture Notes in Computer Science, vol. 8561, pp. 422–429. Springer International Publishing (2014)
24. Woltzenlogel Paleo, B., Delahaye, D.: All about Proofs, Proofs for All. College Publications (to appear in 2015)