Greedy Pebbling: Towards Proof Space Compression

Andreas Fellner * and Bruno Woltzenlogel Paleo **

fellner.a@gmail.com bruno@logic.at
Theory and Logic Group
Institute for Computer Languages
Vienna University of Technology

1 Introduction

Proofs generated by SAT-solvers can be huge. Checking their correctness can not only take a long time but also consume a lot of memory. In an ongoing project for controller synthesis based on the extraction of interpolants from SMT-proofs [15], for example, post-processing a proof takes hours and may reach the limit of memory available today in a single node of a computer cluster (256GB). This issue is even more relevant in application scenarios in which the proof consumer, who is interested in independently checking the correctness of the proof, might have less available memory than the proof producer. This is in part because, while the proof checker reads a usual proof file and checks the proof it contains, every proof node (containing a clause) that is loaded into memory has to be kept there until the end of the whole proof checking process, since the proof checker does not know whether a proof node will still need to be used and re-reading the proof file to reload and recheck proof nodes would be too time-consuming.

To address this issue, recently proposed proof formats such as DRUP [13] and BDRUP [14] allow enriching a proof file with instructions that inform a proof checker when a proof node can be released from memory. Other proof formats, such as the TraceCheck format [3] could also be enriched analogously. Such node deletion instructions can be added by a proof-generating SAT-solver during proof search in the periodic clean-up of its database of derived learned clauses; for every clause the SAT-solver deletes during this phase, this deletion can be recorded in the proof file.

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 checker will need. Therefore, this *deletion-during-proof-postprocessing* approach ought to be seen not as a replacement but rather as an independent complement to the *deletion-during-proof-search* already performed by state-of-the-art proof-generating SAT-solvers.

The new methods proposed here exploit an analogy between proof checking and playing *Pebbling Games* [16, 11]. The particular version of pebbling game

* Supported by the Google Summer of Code 2013 program.
** Supported by the Austrian Science Fund, project P24300.

relevant for proof checking is defined precisely in Section 3 and the analogy to proof checking is explained in detail in Section 4. 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-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). 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 (described in Section 2) satisfies the DAG requirement. The experiments (Section 8) evaluate the proposed algorithms on thousands of SAT- and SMT-proofs.

2 Propositional Resolution Calculus

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

Definition 1 (Proof). *A* proof φ *is a tuple* $\langle V, E, v, \Gamma\rangle$*, such that* $\langle V, E\rangle$ *is a labeled directed acyclic graph whose edges are labeled with literals* $\ell \in \mathcal{L}$*, i.e.* $E \subseteq V \times \mathcal{L} \times V$*,* $v \in V$*,* Γ *is a clause and one of the following holds:*

1. $V = \{v\}, E = \emptyset$
2. *There are proofs* $\varphi_L = \langle V_L, E_L, v_L, \Gamma_L\rangle$ *and* $\varphi_R = \langle V_R, E_R, v_R, \Gamma_R\rangle$ *such that there exists a literal* ℓ *such that* $\overline{\ell} \in \Gamma_L$*,* $\ell \in \Gamma_R$*, and let* $v \notin (V_L \cup V_R)$ *then*

$$V = (V_L \cup V_R) \cup \{v\}$$
$$E = E_L \cup E_R \cup \left\{ v_L \xrightarrow{\overline{\ell}} v, v_R \xrightarrow{\ell} v \right\}$$
$$\Gamma = \left(\Gamma_L \setminus \left\{\overline{\ell}\right\}\right) \cup (\Gamma_R \setminus \{\ell\})$$

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

Note that in case 2 of Definition 1 V_L and V_R are not required to be disjunct. Therefore the underlying structure of a proof is 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 general DAG structure [6, 5]. The reuse of proof nodes plays a central role in proof compression [10]. Also note that a DAG corresponding to a proof has exactly one sink, which is called its root node, by definition.
For example a node of a proof $\langle V, E, v, \Gamma \rangle$ will be meant to be some $s \in V$. From hereon, if free from ambiguity, proofs and their underlying DAGs will not be distinguished.

3 Pebbling Game

Pebbling games are played on directed acyclic graphs and pebbles are placed on nodes following the rules of the game. The goal is to put a pebble on some target node. Pebbling games were introduced in the 1970's to model programming language expressiveness [19, 22] and compiler construction [21]. More recently, pebbling games have been used to investigate various questions in parallel complexity [7] and proof complexity [2, 9, 17]. They are used to obtain bounds for space and time requirements and trade-offs between the two measures [8, 1]. Space requirements are modeled with the number of pebbles used. Time requirements are reflected by the number of rounds played. From hereon *to pebble* means to mark a node with a pebble and *to unpebble* means to remove the mark off a node.

Definition 2 (Bounded Pebbling Game). *The* Bounded Pebbling Game *is played by one player on a DAG $G = (V, E)$ with one distinguished node $s \in V$. The goal of the game is to pebble s, respecting the following rules:*

1. *A node v is pebbleable* iff *all predecessors of v in G are pebbled and v is currently not pebbled.*
2. *Pebbled nodes can be unpebbled at any time.*
3. *Once a node has been unpebbled, it may not be pebbled in a later round.*

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

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

In this work we investigate space requirements when time requirements are fixed. Fixing time is a design choice, see Section 4, and it corresponds to rule 3. Including this rules sets a bound $O(|V|)$ for the number of rounds.

Definition 3 (Strategy). *A* pebbling strategy σ *for the Bounded Pebbling Game, played on a DAG* $G = (V, E)$ *and distinguished node* s*, is a sequence of moves* $(\sigma_1, \ldots, \sigma_n)$ *of the player such that* $\sigma_n = p(s)$*.*

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

Definition 4 (Pebbling number). *The* pebbling number of a pebbling strategy $(\sigma_1, \ldots, \sigma_n)$ *is* $max_{i \in \{1 \ldots n\}} |\{v \in V \mid v \text{ is pebbled in round } i\}|$*. The* pebbling number of a DAG G and node s *is the minimum pebbling number of all pebbling strategies for* G *and* s*.*

Note that Definitions 2 and 3 leave the player freedom when to do unpebbling moves. With the aim of finding strategies with low pebbling numbers, for every unpebbling move there is a canonical round make them, as will be shown in Section 4.

The Bounded Pebbling Game from definition 2 differs from the Black Pebbling Game discussed in [12, 20] 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 [8]. 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 [8]. Deciding whether the pebbling number of a graph G and node s is smaller than k is PSPACE-complete in the absence of rule 3 [11] and NP-complete when rule 3 is included [21].

4 Pebbling and Proof Processing

The problem of processing a proof with minimal memory consumption is analogous to the problem of finding a pebbling strategy with minimal pebbling number. Proof processing could be checking its correctness, manipulating it or extracting information from it. The following definition makes the notion of proof processing formal.

Definition 5 (Proof Processing).

Let φ *be a proof with nodes* V *and* T *be an arbitrary set. A function* $f : V \times T \times T \to T$ *is a* processing function *if there is a function* $g_f : V \to T$ *such that for every* $v \in V$ *with* $P_v^\varphi = \emptyset$ *(i.e.* v *represents an axiom),* $g_f(v) = f(v, t_1, t_2)$ *for all* $\{t_1, t_2\} \subseteq T$*. Let* $\mathcal{F}$ *be the set of processing functions. The* apply function $\mathrm{ap} : V \times \mathcal{F} \to T$ *is defined recursively as follows.*

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

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

Example 1. Checking the correctness of a proof (i.e. checking for the absence of faulty resolution steps) can be checked in terms of the following processing function with $T = \{\top, \bot\}$ and $\wedge$ being the usual boolean and-operation.

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

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

In Section 3 it was pointed out that strategies with minimal pebbling numbers may require to play exponentially many rounds when rule 3 is not used. 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 requiring exponentially many I/O operations and processing steps 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.

In order to process a node, 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. Processing a node and I/O operations are typically more expensive than extra memory consumption, therefore in our setting every node can be processed only once, which corresponds to rule 3. 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 3 makes it impossible to process the whole proof. The optimal moment to remove a node from memory is uniquely determined by the order nodes are processed, see Theorem 1.

Definition 5 does not specify in what 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 6 (Topological Order). *A topological order of a proof φ is a total order relation $\prec$ on V_φ, such that for all $v \in V_\varphi$, 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 $j < i$* iff *$\sigma_j \prec \sigma_i$.* □

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 to order 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 moments 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 pebbling strategies.

Definition 7 (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 recursively:*

$$\sigma_1 = p(v_1)$$
$$\sigma_i = \begin{cases} u(v) \text{ for all } c \in C_v^\varphi \text{ exists } k < i \text{ such that } \sigma_k = p(u) \\ p(v) \text{ otherwise, where } v = min_\prec(w \mid \text{ for all } l < i : \sigma_l \neq p(w)) \end{cases}$$

□

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. Definition 7 prioritizes unpebbling over pebbling moves. Therefore the canonical topological pebbling strategy makes unpebbling moves as soon as possible. Consider the moment for unpebbling an arbitrary node v in the canonical pebbling strategy. Unpebbling it later could only possibly increase the pebble number. To reduce the pebble number, v would have to be unpebbled earlier than some preceding pebbling move. But, by definition of canonical pebbling strategy, the immediately preceding pebbling move pebbles the last child of v w.r.t. $\prec$. Therefore, unpebbling v earlier would make it impossible for its last child to be pebbled later without violating the rules of the game. □

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.

Definition 8 (Space). *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* $\prec$*.* □

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; hence, enumeration is not a feasible option when trying to find a good topological order.

Theorem 2. *There is a sequence of proofs* $(\varphi_1, \ldots, \varphi_m, \ldots)$ *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 $A_\varphi = \{v_{k_1}, \ldots, v_{k_m}\}$, then $(v_{k_1}, \ldots, v_{k_m}, v_{l_1}, \ldots, v_{l_{n-m}})$, where $(l_1, \ldots, l_{n-m}) = (1, \ldots, n) \setminus (k_1, \ldots, k_m)$, 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).

5 Pebbling as a Satisfiability Problem

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

Definition 9 (Pebbling SAT encoding). *The propositional formula obtained by conjuncting the following four constraints expresses the existence of a pebbling strategy for φ with pebbling number smaller or equal k.*

1. *The root is pebbled in the last round*

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

2. *No node is pebbled initially*

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

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

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

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

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

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

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

Theorem 3 (Correctness of pebbling SAT encoding).
$\Psi = \Psi_1 \wedge \Psi_2 \wedge \Psi_3 \wedge \Psi_4$ *is satisfiable* iff *there exists a pebbling strategy using no more than k pebbles*

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

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

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

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

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

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

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

6 Greedy Pebbling Algorithms

Theorem 2 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

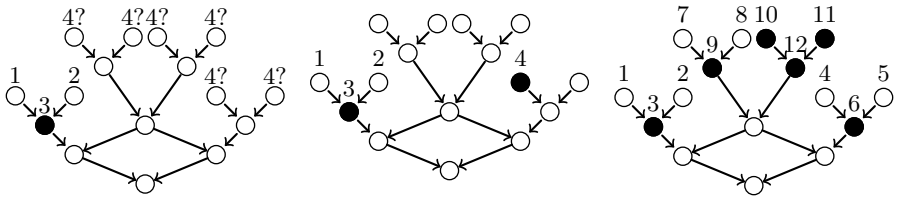

Fig. 1: Top-Down Pebbling

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

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

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

are used are shown in the rightmost picture in Figure 1. However the pebbling number of this proof is 5.

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 r to its axioms. The algorithm constructs the order by visiting nodes and their premises recursively. For every node v the order in which the premises of v are visited is decided heuristically. After visiting the premises, n is added to the current sequence of nodes. Since axioms do not have any premises, there is no recursive call for axioms and these nodes are simply added to the sequence. The recursion is started with the call $BUpebble(\varphi, r, \emptyset, ())$. Since all proof nodes are ancestors of the root, the recursive calls will eventually visit all nodes once and a topological total order will be found. Bottom-Up Pebbling corresponds to the apply function ap(.) defined in Section 4 with the addition of a visit order of the premises. Also previously visited nodes are not visited again.

Algorithm 2: BUpebble

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

```
1 V_1 = V ∪ {v};
2 N = P_v^φ \ V;                    // Only unprocessed premises are visited
3 S_1 = S;
4 while N is not empty do
5 |   choose p ∈ N heuristically; N = N \ p;
6 |   S_1 = S_1 ::: BUpebble(φ, p, V, S);        // ::: is the concatenation of
  |   sequences
7 return S_1 ::: (v);
```

Example 3. Figure 2 shows part of an execution of Bottom-Up Pebbling on the same proof as presented in Figure 1. Nodes chosen by the heuristic, to be processed before the respective other premise, are marked dashed. Suppose that similarly to the Top-Down Pebbling scenario, nodes have been chosen in such a way that the initial pebbling sequence is $(1, 2, 3)$. However, the choice of where to go next is predefined by the dashed nodes. Consider the dashed child of node 3. Since 3 has been completely processed, the other premise of its dashed child is visited next. The result is that the middle subgraph is pebbled while only one external node is pebbled, while it were two in the Top-Down scenario. At no point more than five pebbles will be used for pebbling the root node, which

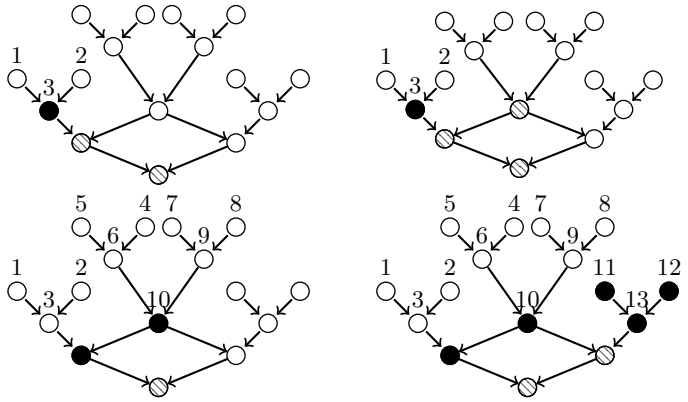

Fig. 2: Bottom-Up Pebbling

is shown in the bottom right picture of the figure. This is independently of the heuristic choices.

6.3 Remarks about Top-Down and Bottom-Up Pebbling

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

The experiments presented in Section 8 show that in practice, Bottom-Up Pebbling performs much better. Example 2 shows two principles that result in pebbling strategies with small pebbling numbers and are likely to be violated by the Top-Down Pebbling algorithm.

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

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

pebbling one subproof can influence the number of pebbles used for another subproof in situations where nodes are share between subproofs. The principle is demonstrated in the following example.

Example 4. Figure 3 shows a simple proof φ with two subproofs φ_0 (left branch) and φ_1 (right branch). As shown in the leftmost diagram, assume $s(\varphi_0, \prec_0) = 4$ and $s(\varphi_1, \prec_1) = 5$, where $\prec_0$ and $\prec_1$ represent some topological order of the respective subproofs with the corresponding pebbling numbers. After pebbling one of the subproofs, the pebble on its root node has to be kept there until the root of the other subproof is also pebbled. Only then the root node can be pebbled. Therefore, $s(\varphi, \prec) = s(\varphi_j, \prec_j)+1$ where $\prec$ is obtained by first pebbling according to $\prec_j$, then by $\prec_{1-j}$ followed by pebbling the root. Choosing to pebble the less spacious subproof φ_0 first results in $s(\varphi, \prec) = 6$, while pebbling the more spacious one first gives $s(\varphi, \prec) = 5$.

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

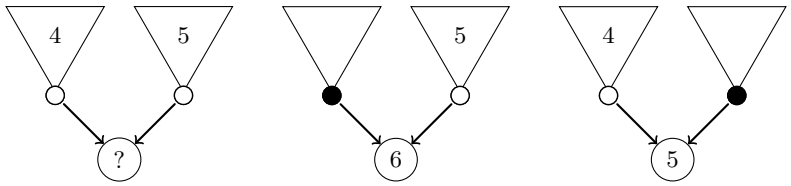

Fig. 3: Spacious subproof first

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

Note that to satisfy the requirement for e_{h_n}, some trivial node evaluation like mapping nodes to their address in memory can be used. In the next chapters

we present heuristics, which are cheap to compute and are justified by relating them to the semantics of the Bounded Pebbling Game. We will not elaborate on effects of reordering the heuristics within full heuristics.

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 4 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, using some topological order $\prec$. Before the traversal, $e_h(v) = 0$ for every node v. During the traversal $e_h(v)$ is incremented by 1, if v is the last child of the currently processed node w.r.t. $\prec$. For this heuristic $S_h = \mathbb{N}$. To some extent, this heuristic is paradoxical: v may be the last child of a node v' according to $\prec$, but pebbling it early may result in another topological order $\prec^*$ according to which v is not the last child of v'. Nevertheless, sometimes the proof structure ensures that some nodes are the last child of another node irrespective of the topological order. An example is shown in Figure 4, where the dashed line denotes a recursive predecessor relationship and the bottommost node is the last child of the top right node in every topological order.

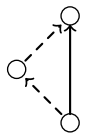

Fig. 4: Bottommost node as necessary last child of right topmost node

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

In Example 2 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. A sphere $K_r^G(v)$ with radius r around the node v in the graph $G = (V, E)$ is the set $\{p \in V \mid v \text{ can be reached from } p \text{ visiting at most } r \text{ edges}\}$, where edges are considered undirected. The heuristic uses the following functions based on the spheres:

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

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

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 with the help of the recursive function rec:

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

8 Experiments

All the pebbling algorithms and heuristics described in the previous sections have been implemented in the hybrid functional and object-oriented programming language Scala (www.scala-lang.org) as part of the Skeptik library for proof compression (github.com/Paradoxika/Skeptik) [18].

To evaluate the algorithms and heuristics, experiments were executed[1] on four disjoint sets of proof benchmarks (Table 1). TraceCheck_1 and TraceCheck_2

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

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

Table 1: Proof benchmark sets

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

Table 2: Experimental results

contain proofs produced by the SAT-solver `PicoSAT` [4] on unsatisfiable benchmarks from the SATLIB (www.satlib.org/benchm.html) library. The proofs[2] are in the TraceCheck proof format, which is one of the three formats accepted at the *Certified Unsat* track of the SAT-Competition. veriT_1 and veriT_2 contain proofs produced by the SMT-solver veriT (www.verit-solver.org) on unsatisfiable problems from the SMT-Lib (www.smtlib.org). These proofs[3] are in a proof format that resembles SMT-Lib's problem format and they were translated into pure resolution proofs by considering every non-resolution inference as an axiom.

Table 2 summarizes the results of the experiments. The two presented algorithms are tested in combination with the four presented heuristics. The Children and LastChild heuristics were tested on all four benchmark sets. The Distance and Decay heuristics were tested on the sets TraceCheck_2 and veriT_2. The relative performance is calculated according to Formula 1, where f is an algorithm

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

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

Table 3: Improvement of LastChild using Decay Heuristic

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.

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

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

Table 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 big.

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

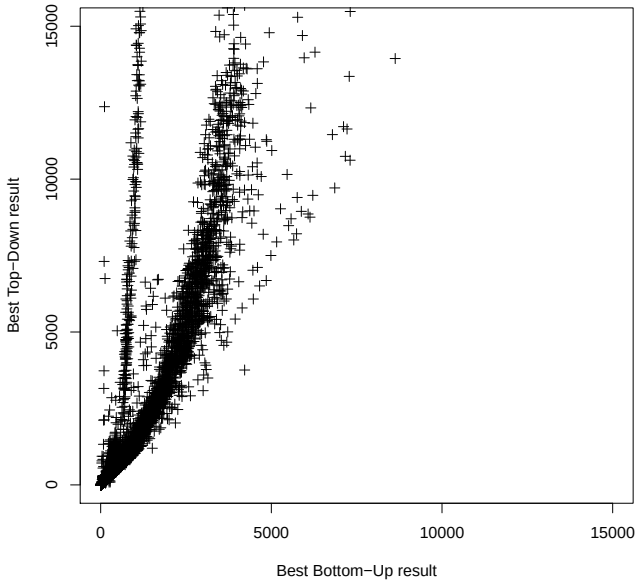

Fig. 5: Space measures of best Bottom-Up and Top-Down result

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

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

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

9 Conclusion

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

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

References

1. Ben-Sasson, E.: Size space tradeoffs for resolution. In: STOC. pp. 457–464 (2002)
2. 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)
3. Biere, A.: Tracecheck resolution proof format (2006), http://fmv.jku.at/tracecheck/README.tracecheck
4. Biere, A.: Picosat essentials. JSAT 4(2-4), 75–97 (2008)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
6. 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)
7. Chan, S.M.: Pebble games and complexity (2013)
8. van Emde Boas, P., van Leeuwen, J.: Move rules and trade-offs in the pebble game. In: Theoretical Computer Science. pp. 101–112 (1979)
9. Esteban, J.L., Torán, J.: Space bounds for resolution. Inf. Comput. 171(1), 84–97 (2001)
10. Fontaine, P., Merz, S., Paleo, B.W.: Compression of propositional resolution proofs via partial regularization. In: CADE. pp. 237–251 (2011)
11. Gilbert, J.R., Lengauer, T., Tarjan, R.E.: The pebbling problem is complete in polynomial space. SIAM J. Comput. 9(3), 513–524 (1980)
12. Hertel, P., Pitassi, T.: Black-white pebbling is pspace-complete. Electronic Colloquium on Computational Complexity (ECCC) 14(044) (2007)
13. Heule, M.: Drup proof format (2007), www.cs.utexas.edu/~marijn/drup/
14. Heule, M.: Bdrup proof format (2013), www.satcompetition.org/2013/certunsat.shtml

15. 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)
16. Kasai, T., Adachi, A., Iwata, S.: Classes of pebble games and complete problems. SIAM J. Comput. 8(4), 574–586 (1979)
17. Nordström, J.: Narrow proofs may be spacious: Separating space and width in resolution. SIAM J. Comput. 39(1), 59–121 (2009)
18. Paleo, B.W., Boudou, J., Fellner, A.: Skeptik system description
19. Pippenger, N.: Comparative schematology and pebbling with auxiliary pushdowns (preliminary version). In: STOC. pp. 351–356 (1980)
20. Pippenger, N.: Advances in pebbling. Springer (1982)
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)