Greedy Pebbling for 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

Abstract. This paper describes algorithms and heuristics for playing a *Pebbling Game*. Playing the game with a small number of pebbles is analogous to checking 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.

1 Introduction

Proofs generated by SAT- and SMT-solvers can be huge. Checking their correctness or extracting information (e.g. 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 [12], for example, extracting an interpolant from an SMT-proof takes hours and reaches 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 end of the whole proof processing, 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 DRUP [10] and BDRUP [11] enrich the older RUP proof format with node deletion instructions. Other proof formats, such as the TraceCheck format [3] or formats for SMT-proofs, 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.

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

The new methods proposed here exploit an analogy between proof checking and playing *Pebbling Games* [13, 8]. The particular version of pebbling game 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- 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). 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 $\mathcal{L}$. 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 directed acyclic graph whose edges are labeled with literals $\ell \in \mathcal{L}$, $v \in V$, 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*

$$\begin{aligned} & v \textit{ is a new node} \\ & V = V_L \cup V_R \cup \{v\} \\ & E = E_L \cup E_R \cup \left\{ v_L \xrightarrow{\overline{\ell}} v, v_R \xrightarrow{\ell} v \right\} \\ & \Gamma = \left(\Gamma_L \setminus \{\overline{\ell}\}\right) \cup \left(\Gamma_R \setminus \{\ell\}\right) \end{aligned}$$

v is called the root *of φ and Γ its* conclusion.
In case 2 φ_L and φ_R are called premises *of φ and φ is called a* child *of both φ_L and φ_R. The transitive closure of the premise relation is the* subproof *relation. A subproof which has no premises is an* axiom *of the respective proof. 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 $l(\varphi)$.* □

Note that in case 2 of definition 1 V_L and V_R are not required to be disjoint. Therefore the underlying structure of proofs are DAGs as opposed to simple trees. Modern SAT- and SMT-solvers produce proofs of such kind [**?**,**?**] and the reuse of proof nodes plays a central role in proof compression [**?**].
Also note that a DAG corresponding to a proof by definition has exactly one sink, also called its root node.
From hereon, if free from ambiguity, proofs and their underlying DAGs will not be distinguished. For example a node of a proof $\langle V, E, v, \Gamma \rangle$ will be meant to be some $s \in V$.

3 Pebbling Game

Pebbling games were introduced in the 1970's to model programming language expressiveness [15, 18] and compiler construction [17]. More recently, pebbling games have been used to investigate various questions in parallel complexity [5] and proof complexity [1, 7, 14]. They are used to obtain bounds for space and time requirements and trade-offs between the two measures [6, **?**]. In this paper we investigate space requirements when time is fixed.
To pebble a node is to put a pebble on it; *to unpebble* is to remove a pebble; a node is *pebbleable* if it is not pebbled but can be pebbled.

Definition 2 (Static Pebbling Game). *The* Static 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.*
2. *Nodes can be unpebbled at any time.*
3. *Once a node has been unpebbled, it can not be pebbled again later.*

□

Note that as a consequence of rule 1, pebbles can be put on nodes without predecessors at any time.
Also note that rule 3 corresponds to fixing the time required to play the game, as with this rule the number of moves will be $O(n)$.

Definition 3 (Strategy). *A* pebbling strategy *for the Static Pebbling Game, played on a DAG $G = (V, E)$ and distinguished node s, is a sequence of moves $(m_1, \ldots, m_n)$, such that the following formula corresponding to the rules and the goal of the game, using the predicates* P(.) *and* U(.)*, is true:*

$$m_n = \mathrm{P}(s) \ \wedge$$
$$\Big(\forall i {\in} \{1 \ldots n\} : m_i = \mathrm{P}(v) \Rightarrow$$
$$\big(\forall k {\in} \{i+1 \ldots n\} : m_k \neq \mathrm{P}(v) \wedge \forall o \in P_v^G : \mathrm{pebbledAt}(i, o, m_1^n)\big)\Big) \ \wedge$$
$$\big(\forall i {\in} \{1 \ldots n\} : m_i = \mathrm{U}(v) \Rightarrow \mathrm{pebbledAt}(i, v, m_1^n)\big)$$

where $\mathrm{pebbledAt}(i, v, m_1^n) := \exists j {\in} \{1 \ldots i-1\} : (m_j = \mathrm{P}(v) \wedge \forall l {\in} \{j \ldots i-1\} : m_l \neq \mathrm{U}(v))$

In the formula above P_v^G denotes, similarly as defined for proofs, the set of predecessors of v in G. The following definition allows to measure how many pebbles are required to play the Static Pebbling Game on a given graph.

Definition 4 (Pebbling number). *The* pebbling number of a pebbling strategy $(m_1, \ldots, m_n)$ *is defined as*

$$max_{i \in \{1 \ldots n\}}(|\{v \in V \mid \mathrm{pebbledAt}(i, v, m_1^n)\}|)$$

where $\mathrm{pebbledAt}(i, v, m_1^n)$ *denotes the same formula as in definition 3.*
The pebbling number of a DAG G and node s *is defined as the minimum pebbling number of all pebbling strategies for G and s.*

Note that the definitions 2 and 3 leave freedom when to do unpebbling moves. With the aim of finding strategies with low pebbling numbers, there is a canonical way when to do these moves, as will be shown later.
The *Static Pebbling Game* from definition 2 slightly differs from the Black Pebbling Game discussed in [9, 16] in two aspects. Firstly, the Black Pebbling Game does not include rule 3. Not having this rule allows for pebbling strategies with lower pebbling numbers ([17] has an example on page 1), but at a possible cost of exponentially more moves [6]. 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 when allowing to moving pebbles are discussed in [6]. 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 [8] and NP-complete when rule 3 is included [17].

4 Pebbling and Proof Checking

The problem of checking the correctness of a proof while minimizing the memory consumption is analogous to the problem of playing the pebbling game on the proof while minimizing the number of pebbles. Checking the correctness of a node and storing it in memory corresponds to pebbling it. Deleting a node from memory corresponds to unpebbling it. In order to check the correctness of a node, its premises must have been checked before and must still be stored in memory (rule 1 of the pebbling game). A node that has already been checked can be removed from memory at any time (rule 2). The correctness of a node should be checked only once (rule 3).

Proof files generated by SAT- and SMT-solvers usually already list the nodes of a proof in a topological order. Even if this were not the case, it is simple to generate a topological order by traversing the proof once.

Definition 5 (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 pebbling strategy $(m_1, \ldots, m_n)$ respects *a topological order $\prec$ iff*

$$\forall i {\in \{1 \ldots n\}} : m_i = \mathrm{P}(v) \Rightarrow (\forall j {\in \{1 \ldots i-1\}} : m_j = \mathrm{P}(o) \Rightarrow o \prec v)$$

□

A topological order $\prec$ can be represented by a sequence $(v_1, \ldots, v_n)$ of proof nodes, by defining $\prec := \{(v_i, v_j) \mid 1 \leq i < j \leq n\}$. This sequence can be interpreted as a particular pebbling strategy, played on the DAG that corresponds to the proof, that pebbles nodes according to the topological. As already briefly noted in section 3, unpebbling moves are given implicitly by what we call the canonical topological pebbling strategy.

Definition 6 (Canonical Topological Pebbling Strategy). *The* canonical topological pebbling strategy *for a proof φ, its root node s and a topological order $\prec$, represented as a sequence $(v_1, \ldots, v_n)$ is defined as follows:*

$$m_1 = \mathrm{P}(v_1); k := 0$$

$$m_i = \begin{cases} \mathrm{U}(v_{i-k}) \textit{ iff } \forall o \in P^{\varphi}_{v_{i-k}} : \mathrm{pebbledAt}(i-k, o, m_1^{i-1}); k := k+1 \\ \mathrm{P}(v_{i-k}) \textit{ otherwise} \end{cases}$$

where $\mathrm{pebbledAt}(i-k, o, m_1^{i-1})$ *denotes the same formula as in definition 3.* □

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

Proof. (Sketch) All the pebbling strategies respecting $\prec$ differ only w.r.t. their unpebbling moves. Consider the unpebbling of 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. □

Theorem 1 shows that, in the version of the pebbling game considered here, the problem of finding a strategy with a low pebble number can be reduced to the problem of finding a topological order whose canonical strategy has a low pebble number. Unpebbling moves can be omitted, because the optimal moment to unpebble a node is immediately after its last child has been pebbled.

Assuming that nodes are approximately of the same size, the maximum memory needed to check a proof is proportional to the maximum number of nodes that have to be kept in memory while checking the proof according to a given topological order. Thus memory consumption can be estimated by the following measure:

Definition 7 (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. Let φ be a proof with nodes $v_1, \ldots, v_n$ and let v_n be its the root node. Due to rule 3 of the Static 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 find a strategy. For every $x \in \{1, \ldots, k\}$, every $j \in \{1, \ldots, n\}$ and every $t \in \{1, \ldots, n\}$ there is a propositional variable $p_{x,j,t}$, denoting if that pebble x is on node v_j at the time of the pebbling move t. The following constraints, combined conjunctively, are satisfiable *iff* there is a pebbling strategy for φ, using at most k pebbles. If satisfiable, a pebbling strategy can be read off from any satisfying assignment.

1. The root is pebbled at the time of the last move

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

2. At most one node is pebbled initially

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

3. At least one axiom is pebbled initially

$$\bigvee_{x=1}^{k} \bigvee_{j \in A_\varphi} p_{x,j,1}$$

4. A pebble can only be on one node

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

5. For pebbling a node, its premises have to be pebbled and only one node is pebbled each move

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

The sets A_{φ} and P_j^{φ} are interpreted as sets of indices of the respective nodes. This encoding is polynomial, both in n and k. However constraint 5 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 5 alone. Therefore, although theoretically possible to play the pebbling game via SAT-solving, this is practically infeasible for compressing proof space.

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 a better though not necessarily optimal topological order. They are both parameterized by the same heuristics described in Section 7, but differ from each other 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 pebbling game. A human would look at the nodes that are available for pebbling at a given state, 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.

```
   Input: a proof $\varphi$
   Output: A sequence of nodes $S$ representing a topological order $\prec$ of $\varphi$
1  $S = ()$; // the empty sequence
2  $N = A_\varphi$; // initialize pebbleable nodes with Axioms
3  while $N$ is not empty do
4      choose $v \in N$ heuristically;
5      for each $c \in C_v^\varphi$ do // check whether $c$ is pebbleable after pebbling $v$
6          if $\forall p \in P_c^\varphi : p \in S$ then
7              $N = N \cup \{c\}$;
8      $N = N \setminus \{v\}$;
9      $S = S ::: (v)$; // ::: is the concatenation of sequences
10 return $S$;
```

Algorithm 1: `Top-Down Pebbling`

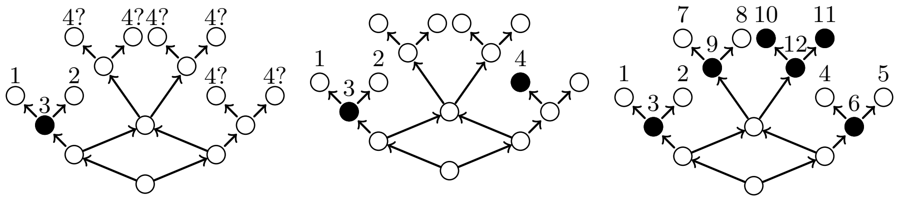

Fig. 1: Top-Down Pebbling

Example 1. Top-Down Pebbling often ends up finding a sub-optimal pebbling strategy regardless of the heuristic used. 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.

6.2 Bottom-Up Pebbling

`Bottom-Up` Pebbling (Algorithms 2 and 3) constructs a topological order of a proof φ while traversing it from its root node to its axioms. The algorithm constructs the order by visiting nodes and their premises recursively. At 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 by a call to visit the root. 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.

```
   Input: a proof $\varphi$ with root node $r$
   Output: A sequence of nodes $S$ representing a topological order $\prec$ of $\varphi$
1  $S = ()$; // the empty sequence
2  $V = \emptyset$;
3  return visit($\varphi$,$r$,$V$,$S$);
```

Algorithm 2: `Bottom-Up Pebbling`

```
   Input: a proof $\varphi$
   Input: a node $v$
   Input: a set of visited nodes $V$
   Input: initial sequence of nodes $S$
   Output: a sequence of nodes
1  $V_1 = V \cup \{v\}$;
2  $N = P^{\varphi}_{v} \setminus V$;
3  $S_1 = S$
4  while $N$ is not empty do
5      choose $p \in N$ heuristically;
6      $N = N \setminus p$;
7      $S_1 = S_1 ::: visit(\varphi, p, V, S)$; // ::: is the concatenation of sequences
8  return $S_1 ::: (v)$;
```

Algorithm 3: `visit`

Example 2. Figure 2 shows part of an execution of `Bottom-Up` Pebbling on the same graph as presented in Figure 1. Nodes chosen by the heuristic, during the Bottom-Up traversal, to be processed before the respective other premise are marked dashed. 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 and pebbled, the other premise of its dashed child is visited next. The result is that node the middle subgraph is pebbled while only one external node is pebbled, while it have been two in the Top-Down scenario. At no point more than five pebbles will be used for pebbling the root node, which is shown in the bottom right picture of the figure. This is independently of the heuristic choices.
Not that this example underlines the point of two observations: pebbling choices should be made local and hard subgraphs should be pebbled early.

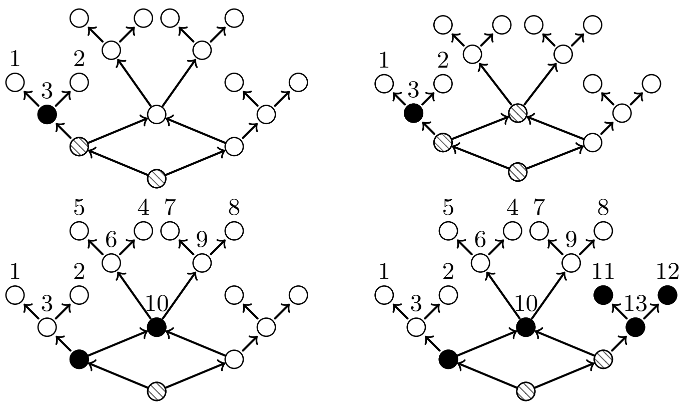

Fig. 2: Bottom-Up Pebbling

6.3 Remarks about Top-Down and Bottom-Up Pebbling

In principle every topological order of a given proof can be constructed using Top-down or Bottom-up Pebbling. Both algorithms traverse the proof only once and have linear run-time in the proof length (assuming that the heuristic choice requires constant time). Example 1 shows a situation where `Top-Down` Pebbling may pebble a node that is far away from the previously pebbled nodes. This results in a sub-optimal pebbling strategy. As discussed in Example 2, `Bottom-Up` Pebbling is more immune to this non-locality issue, because queuing up the processing of premises enforces local pebbling. This suggests that `Bottom-Up` is better than `Top-Down`, which is confirmed by the experiments in Section 8.

7 Heuristics

Both pebbling algorithms described in the previous section have to choose a node $h(N)$ out of a set N at some point, where h is a *heuristic* function. For `Top-Down` Pebbling, N is the set of pebbleable nodes, and for `Bottom-Up` Pebbling, N is the set of premises of a node. Every heuristic h considered here uses an auxiliary *node evaluation* function e_h that maps nodes to elements of a totally ordered set S_h. The chosen node $h(N)$ is then simply defined as $argmax_{v \in N} e_h(v)$ (i.e. $h(N)$ is a node in N such that for all nodes v in N, $e_h(h(N)) \geq e_h(v)$).

7.1 Number of Children Heuristic ("*Ch*")

This heuristic uses the number of children of a node v: $e_h(v) = |C_v^{\varphi}|$ 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 3 shows why it is a good idea to process spacious subproofs first.

Example 3. Figure 3 shows a simple proof φ with two subproofs φ_1 (left branch) and φ_2 (right branch). As shown in the leftmost diagram, assume $s(\varphi_1, <_T^1) = 4$ and $s(\varphi_2, <_T^2) = 5$. 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, $s(\varphi, \prec) = s(\varphi_j, <_T^j) + 1$ where $\prec = <_T^i ::: <_T^j$ and i and j are the indexes of the subproofs pebbled, respectively first and last. Choosing to pebble the least spacious subproof φ_1 first results in $s(\varphi, \prec) = 6$, while pebbling the most spacious one first gives $s(\varphi, \prec) = 5$. This is a simplified situation. The two subproofs do not share nodes. Pebbling one of them does not influence the pebble number of the other.

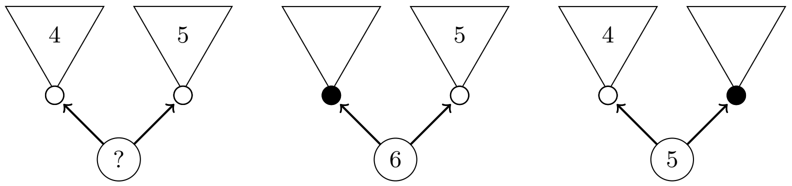

Fig. 3: Spacious subproof first

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 can be used for a heuristic that prefers 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, $e_h(v)$ is set to 0 for every node v. During the traversal $e_h(v)$ is increased 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 $<_T^*$ 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.

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

In Example 1 and Section 6.3 it has been noted that `Top-Down` Pebbling may perform badly if nodes that are far apart are selected. The Node `Distance`

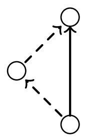

Fig. 4: Bottommost node as necessary last child of right topmost node

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 $\{p \in V \mid \text{ there are at most } r \text{ edges between } p \text{ and } v\}$. The direction of edges is not considered. The heuristic uses the following functions based on the spheres:

$$\begin{aligned} d(v) &:= -min\{r \mid K_r^G(v) \text{ contains a pebbled node}\} \\ s(v) &:= |K_{-d(v)}^G(v)| \\ l(v) &:= max_{<_N} K_{-d(v)}^G(v) \\ e_h(v) &:= (d(v), s(v), l(v)) \end{aligned}$$

where $<_N$ denotes the total order on the initial sequence of pebbled nodes N, i.e. nodes in spheres that were pebbled later are preferred. 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 $<_N$ on N. The spheres $K_r(v)$ can grow exponentially in r. Therefore the maximum radius has to be limited and if no pebbled node is found within in any sphere with this radius, another heuristic has to be used.

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:

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

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) [4]. In order to evaluate them, experiments were executed[1] on four disjoint sets of proof benchmarks (Table 1). Two of them contain proofs produced by the SAT-solver PicoSAT [2] on unsatisfiable benchmarks from the SATLIB (www.satlib.org/benchm.html) library. The proofs[2] are in the TraceCheck format, which is one of the three formats accepted at the *Certified Unsat* track of the SAT-Competition. The other two benchmark sets 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.

Figure 5 relates for each proof the smallest space measure obtained by all algorithms tested on the respective proof and its length in number of nodes. Note that the y-axis, showing the space measures, scale is lower by a factor of 100 than the scale of the x-axis, which displays the proof lengths. 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 checking a proof.

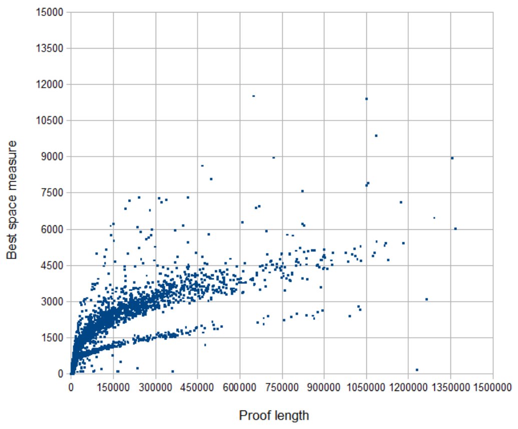

Fig. 5: Best space measure compared to proof length

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

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

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

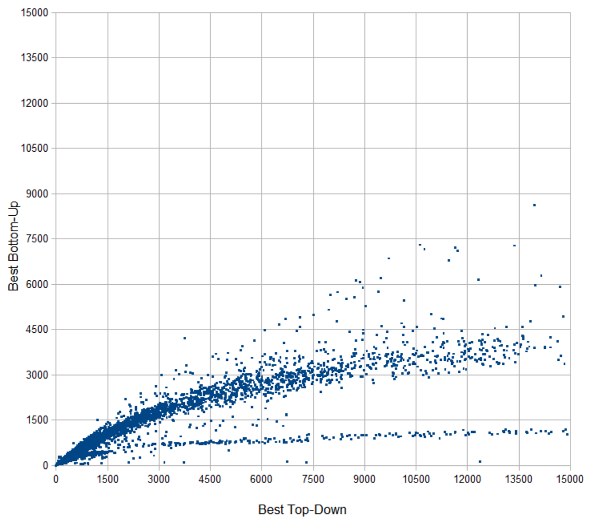

Fig. 6: Spaces obtained with best `Bottom-Up` and `Top-Down` heuristics

Name	Number of proofs	Maximum length	Average length
TRC1	2239	90756	5423
TRC2	215	1768249	268863
SMT1	4187	2241042	103162
SMT2	914	120075	5391

Table 1: Proof benchmark sets

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

Table 2 shows that `Bottom-Up` algorithms construct topological orders with much smaller space measures than `Top-Down` algorithms. This fact is visualized in Figure 6, where each dot represents a proof φ and the x and y coordinates show the space of φ with the topological orders found by, respectively, the best `Top-Down` and `Bottom-Up` algorithms for φ. Some other heuristics (not described in this paper) aimed at improving `Top-Down` Pebbling were tested on small benchmark sets, but none showed promising results.

Furthermore, `Bottom-Up` algorithms are also much faster, as can be seen in the last column of Table 2. This is so because they require fewer comparisons in their heuristic choices. For `Bottom-Up` algorithms, 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). On the other hand, the set N of

Algorithm	**Relative Performance** (%)				**Speed**
Heuristic:Method	**SMT1**	**SMT2**	**TRC1**	**TRC2**	(nodes/ms)
Ch:BU	19,53	-15,79	20,48	**88,57**	**88,55**
Ch:TD	**-22,07**	8,29	-48,33	-67,12	0,30
Lc:BU	**23,42**	36,69	21,47	88,55	84,43
Lc:TD	-20,88	14,20	-64,07	**-110,00**	1,87
Dist(1):BU		-15,72	19,74		21,23
Dist(1):TD		-67,52	-71,21		0,63
Dist(3):BU		-50,27	19,95		0,54
Dist(3):TD		**-74,90**	**-74,09**		**0,08**
Dc(LC,0.5,1,avg):BU		37,39	21,83		47,70
Dc(LC,0.5,7,avg):BU		37,78	22,05		14,01
Dc(LC,3,1,avg):BU		36,86	22,02		63,97
Dc(LC,3,7,avg):BU		34,69	**22,55**		15,31
Dc(LC,0.5,1,max):BU		37,31	21,76		47,03
Dc(LC,0.5,7,max):BU		37,89	21,94		15,26
Dc(LC,3,1,max):BU		37,33	21,79		64,43
Dc(LC,3,7,max):BU		**37,96**	22,13		15,34

Table 2: Experimental results

currently pebbleable nodes, from which `Top-Down` algorithms must choose, is large (e.g. for a perfect binary tree with $2n-1$ nodes, initially $|N| = n$). 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.

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

As expected, the `Decay` Heuristic does improve the results of the underlying heuristic. Note that because of the relative nature of the performance measure and the poor performance of the `Top-Down` algorithms, small performance differences can still be significant. Nevertheless, the performance improvement comes at a high cost in speed.

9 Conclusions

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 simpler and more straightforward `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 subproofs of learned clauses generated by conflict graph analysis.

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., Nordstrom, J.: Short proofs may be spacious: An optimal separation of space and length in resolution. In: Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on. pp. 709–718. IEEE (2008)
2. Biere, A.: Picosat essentials. Journal on Satisfiability, Boolean Modeling and Computation (JSAT p. 2008
3. Biere, A.: Tracecheck resolution proof format (2006), http://fmv.jku.at/tracecheck/README.tracecheck
4. Boudou, J., Fellner, A., Woltzenlogel Paleo, B.: Skeptik [system description]. In: submitted (2014)
5. Chan, S.M.: Pebble games and complexity (2013)
6. van Emde Boas, P., van Leeuwen, J.: Move rules and trade-offs in the pebble game. In: Theoretical Computer Science 4th GI Conference. pp. 101–112. Springer (1979)
7. Esteban, J.L., Torn, J.: Space bounds for resolution. Information and Computation 171(1), 84 – 97 (2001), http://www.sciencedirect.com/science/article/pii/S0890540101929219
8. Gilbert, J.R., Lengauer, T., Tarjan, R.E.: The pebbling problem is complete in polynomial space. SIAM Journal on Computing 9(3), 513–524 (1980)
9. Hertel, P., Pitassi, T.: Black-white pebbling is pspace-complete. In: Electronic Colloquium on Computational Complexity (ECCC). vol. 14 (2007)
10. Heule, M.: Drup proof format (2007), www.cs.utexas.edu/˜marijn/drup/
11. Heule, M.: Bdrup proof format (2013), www.satcompetition.org/2013/certunsat.shtml
12. Hofferek, G., Gupta, A., Könighofer, B., Jiang, J.H.R., Bloem, R.: Synthesizing multiple boolean functions using interpolation on a single proof. In: FMCAD. pp. 77–84. IEEE (2013)
13. Kasai, T., Adachi, A., Iwata, S.: Classes of pebble games and complete problems. SIAM Journal on Computing 8(4), 574–586 (1979)
14. Nordström, J.: Narrow proofs may be spacious: Separating space and width in resolution. SIAM Journal on Computing 39(1), 59–121 (2009)
15. Paterson, M.S., Hewitt, C.E.: Comparative schematology. In: Record of the Project MAC Conference on Concurrent Systems and Parallel Computation. pp. 119–127. ACM (1970)
16. Pippenger, N.: Advances in pebbling. Springer (1982)
17. Sethi, R.: Complete register allocation problems. SIAM journal on Computing 4(3), 226–248 (1975)
18. Walker, S., Strong, H.: Characterizations of flowchartable recursions. Journal of Computer and System Sciences 7(4), 404 – 447 (1973), http://www.sciencedirect.com/science/article/pii/S0022000073800327