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

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 and the proposed pebbling algorithms are 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-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 [?], 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 and BDRUP [?,?] 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 [?] could also be enriched analogously. Such node deletion instructions are currently added by proof-generating sat-solvers 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.

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

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 methods proposed here exploit an analogy between proof checking and playing *Pebbling Games* [7, 5]. The particular version of pebbling game relevant for proof checking is defined precisely in Section **??** and the analogy to proof checking is explained in detail in Section **??**. The proposed pebbling algorithms are greedy (Section **??**) and based on heuristics (Section **??**). As discussed in Sections **??** and **??**, approaches based on exhaustive enumeration or 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, and they could be applied to proofs represented in any proof 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 **??**) satisfies the DAG requirement. The experiments (Section **??**) 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 directed acyclic graph $\langle V, E, \Gamma \rangle$, where V is a set of nodes and E is a set of edges labeled by literals (i.e. $E \subset V \times \mathcal{L} \times V$ and $v_1 \xrightarrow{\ell} v_2$ denotes an edge from node v_1 to node v_2 labeled by ℓ), is a proof of a clause Γ iff it is inductively constructible according to the following cases:*

1. *If Γ is a clause, $\widehat{\Gamma}$ denotes some proof $\langle \{v\}, \emptyset, \Gamma \rangle$, where v is a new node.*
2. *If ψ_L is a proof $\langle V_L, E_L, \Gamma_L \rangle$ and ψ_R is a proof $\langle V_R, E_R, \Gamma_R \rangle$ and ℓ is a literal such that $\overline{\ell} \in \Gamma_L$ and $\ell \in \Gamma_R$, then $\psi_L \odot_\ell \psi_R$ denotes a proof $\langle V, E, \Gamma \rangle$ s.t.*

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

where v is a new node and $\rho(\varphi)$ denotes the root node of φ. □

If $\psi = \varphi_L \odot_\ell \varphi_R$, then φ_L and φ_R are *direct subproofs* of ψ and ψ is a *child* of φ_L and φ_R. The transitive closure of the direct subproof relation is the *subproof* relation. A subproof which has no direct subproof is an *axiom* of the proof. V_ψ, E_ψ, A_ψ and Γ_ψ denote, respectively, the nodes, edges, axioms and proved clause (conclusion) of ψ. P_v^ψ denotes the premises and C_v^ψ the children of a node v in a proof ψ. When a proof is represented graphically, the root is drawn at the bottom and the axioms at the top.

3 Pebbling Game

Pebbling games were introduced in the 1970's to model programming language expressivity [9, 12] and compiler construction [11]. More recently, pebbling games have been used to investigate various questions in parallel complexity [2] and proof complexity [1, 4, 8]. They are used to get bounds for space and time requirements and tradeoffs between the two measures [3]. The *pebbling game* from definition 2 is a slight variation of the Black Pebbling Game presented in [6, 10]. *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.

ToDo: briefly explain the difference...

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

1. *If all predecessors of a node v are pebbled, then v is pebbleable.*
2. *Nodes can be unpebbled at any time.*
3. *Each node can be pebbled only once.*

As a consequence of rule 1, pebbles can be put on nodes without predecessors at any time. A pebbling strategy *for G and node s is a sequence of moves in the pebbling game, where the last move pebbles s. The* pebble number of a pebbling strategy *is the maximum number of pebbles that are placed on nodes simultaneously, following the moves of the strategy. The* pebble number of a graph G and node *s is the minimum pebble number of all pebble strategies, for the pebbling game played on G and s.* □

The Black Pebbling Game defined in [6, 10] introduces another rule according to which a pebble can be moved from a predecessor to the node instead of using a fresh one. Including this rule results in strategies that use exactly one pebble less, as shown in [3]. Omitting rule 3 allows pebbling strategies with lower pebbling numbers ([11] has an example on page 1). However, this possibly has a cost of exponentially more moves in the game [3]. Deciding the question whether the pebbling number of a graph G and node s is smaller than k is PSPACE-complete in the absence of rule 3 [5] and NP-complete when rule 3 is included [11].

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 though memory-consuming to generate a topological order by traversing the proof once.

Definition 3 (Topological Order). *A topological order of a proof ψ is a total order relation $<_T$ on V_ψ, such that for all $v \in V_\psi$, for all $p \in P_v^\psi, p <_T v$* □

A topological order $<_T$ can be represented by a sequence $(v_1, \ldots, v_n)$ of proof nodes, by defining $<_T := \{(v_i, v_j) \mid 1 \leq i < j \leq n\}$. This sequence can be interpreted as a particular pebbling strategy that pebbles nodes according to the topological order and unpebbles a node v soon after all its last child is pebbled. This is formally defined below.

Definition 4 (Canonical Topological Pebbling Strategy). *The* canonical topological pebbling strategy *$S(\psi, \rho(\psi), <_T)$ for a DAG ψ and node $\rho(\psi)$ w.r.t. a topological order $<_T$ represented as a sequence $(v_1, \ldots, v_n)$ is defined recursively:*

$$S(\psi, \rho(\psi), t) = \begin{cases} () & , \text{if } t = () \\ pebble(v) :: (U(v, \psi, t) ::: S(\psi, \rho(\psi), r) & , \text{if } t = v :: r \end{cases}$$

$$U(v, \psi, t) = (unpebble(v_p) \mid v_p \in P_v^\psi \text{ and } v_c \leq_T v \text{ forall } v_c \in C_{v_p}^\psi)$$

where :: *is the* cons *list constructor and* ::: *is the list concatenation operator.* □

Theorem 1. *$S(\psi, \rho(\psi), <_T)$ has the minimum pebbling number among all pebbling strategies that pebble nodes according to the topological order $<_T$.*

Proof. (Sketch) All the pebbling strategies respecting $<_T$ differ only w.r.t. their unpebbling moves. Consider the unpebbling of an arbitrary node v in the canonical strategy $S(\psi, \rho(\psi), <_T)$. 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 strategy, the immediately preceding pebbling move pebbles the last child of v. 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 5 (Space). *The* space $s(\psi, <_T)$ *of a proof* ψ *and a topological order* $<_T$ *is the pebbling number of the canonical topological pebbling strategy* $S(\psi, \rho(\psi), <_T)$. □

The problem of compressing the space of a proof ψ and a topological order $<_T$ is the problem of finding another topological order $<'_T$ such that $s(\psi, <'_T) < s(\psi, <_T)$. 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* $(\psi_1, \ldots, \psi_m, \ldots)$ *such that* $length(\psi_m) \in O(m)$ *and* $|T(\psi_m)| \in \Omega(m!)$*, where* $T(\psi_m)$ *is the set of possible topological orders for* ψ_m*.*

Proof. Let ψ_m be a perfect binary tree with m axioms. Clearly, $length(\psi_m) = 2m-1$. Let $(s_1, \ldots, s_n)$ be a topological order for ψ_m. Let $A_\psi = \{s_{k_1}, \ldots, s_{k_m}\}$, then $(s_{k_1}, \ldots, s_{k_m}, s_{l_1}, \ldots, s_{l_{n-m}})$, where $(l_1, \ldots, l_{n-m}) = (1, \ldots, n) \setminus (k_1, \ldots, k_m)$, is a topological order as well. Likewise, $(s_{\pi(k_1)}, \ldots, s_{\pi(k_m)}, s_{l_1}, \ldots, s_{l_{n-m}})$ 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).

Use v instead of s for nodes here

5 Pebbling as a Satisfiability Problem

To find the pebble number of a proof, the question whether this proof can be pebbled using no more than k pebbles, can be encoded as a propositional satisfiability problem. Let ψ be a proof with nodes $v_1, \ldots, v_n$ and let $v_n = \rho(\psi)$. Due to rule 3 of the pebbling game, the number of pebbling moves is exactly n. 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 ψ, 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

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

3. At least one axiom is pebbled initially

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

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

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

The sets A_ψ and P_j^ψ 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 or compress proof space via sat-solving, this is practically infeasible.

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 ψ by traversing it from the axioms to the root node $\rho(\psi)$. 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 P, initialized as A_ψ. When a node v is pebbled, it is removed from P and added to the sequence representing the topological order. The children of v that become pebbleable are added to P. When P becomes empty, all nodes have been pebbled once and a topological order has been found.

Input: a proof ψ
Output: a topological order $<_T$ of ψ represented by a sequence of nodes
$S = ()$; // the empty sequence
$P = A_\psi$;
while P *is not empty* **do**
    choose $v \in P$ heuristically;
    for each $c \in C_v^\psi$ **do**
        if $\forall p \in P_c^\psi : p \in S$ **then**
            $P = P \cup \{c\}$;
    $P = P \setminus \{v\}$;
    $S = S ::: (v)$; // $:::$ is the concatenation of sequences
return S;

Algorithm 1: `Top-Down Pebbling`

Example 1. It is easy to see how top-down pebbling may end up finding a sup-optimal pebbling strategy. Consider the graph shown in figure 3 and suppose that top-down pebbling has already pebbled the initial sequence of nodes $(1, 2, 3)$. For a greedy heuristic that only has information about pebbled nodes, their premises and children, all nodes marked with '4?' are considered equally worthy to pebble next. Suppose the node marked with '4' in the middle graph is chosen and pebbled next. Subsequently, pebbling '5' opens up the possibility to remove a pebble in the next move, which is done by pebbling '6'. After that only '7' and '8' are pebbleable. In this situation, it does not matter which is pebbled first. After pebbling '7' and '8', four pebbles are used, which is one more than what an optimal strategy needs.

6.2 Bottom-Up Pebbling

Bottom-Up Pebbling (Algorithms 2 and 3) constructs a topological order of a proof ψ while traversing it from its root node $\rho(\psi)$ to its axioms. The algorithm constructs the order by visiting nodes and their premises recursively. At every

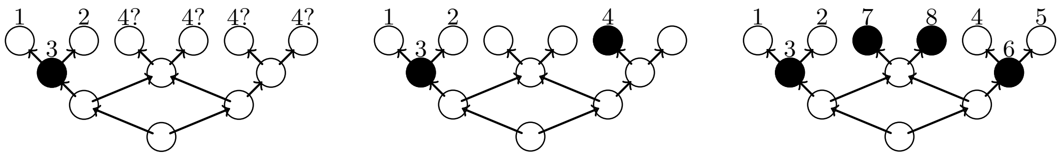

Fig. 1: Top-Down Pebbling

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 ψ
Output: a topological order $<_T$ of ψ represented by a sequence of nodes

$S = ()$; // the empty sequence
$V = \emptyset$;
return `visit`(ψ,$\rho(\psi)$,V,S);

Algorithm 2: `Bottom-Up Pebbling`

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

$V_1 = V \cup \{v\}$;
$D = P_v^\psi \setminus V$;
$S_1 = S$
while D *is not empty* **do**
    choose $p \in D$ heuristically;
    $D = D \setminus p$;
    $S_1 = S_1 ::: \mathit{visit}(\psi, p, V, S)$; // ::: is the concatenation of sequences
return $S_1 ::: (v)$;

Algorithm 3: `visit`

Example 2. Figure 2 shows part of an execution of Bottom-Up Pebbling on the same graph of Figure 3. Nodes chosen by the heuristic, during the bottom-up traversal, to be processed before the respective other premise are marked in gray.

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 gray nodes. Consider the gray child of node '3'. Since '3' has been completely processed and pebbled, the other premise of its gray child is visited next. The result is that node '7' is pebbled earlier and at no point more than 3 pebbles will be used for pebbling the root node. This is so independently of the heuristic choices.

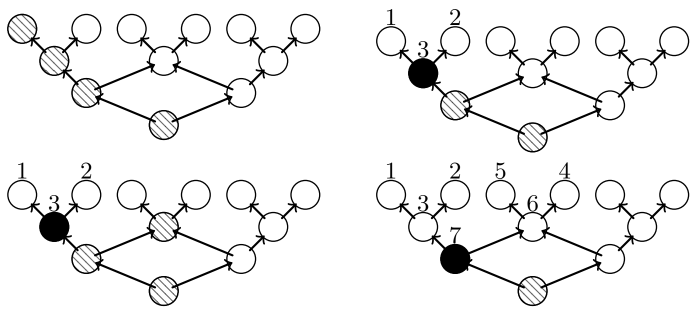

Fig. 2: Bottom-Up Pebbling

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.

Theoretically, can we prove that BUP is always better than TDP if they both use the same heuristics? Experimentally, does it ever happen that TDP is better than BUP? We could make a scatter plot with compression ratios of TDP and BUP on the y and x axis to investigate this...

7 Heuristics

Pebbling heuristics for a proof ψ are defined by a function $h : V_\psi \to H$, where $(H, \prec)$ is a totally ordered set. The pebbling algorithms select a node v out of a set of nodes $N \subseteq V_\psi$ where $v = max_{n \in N} h(n)$ using the order $\prec$. 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.

7.1 Number of Children Heuristic

This heuristic uses the number of children of a node v: $h(v) = |C_v^\psi|$, $H = \mathbb{N}$ and $\prec$ is the *smaller than* relation on $\mathbb{N}$. The intuitive motivation for this heuristic is that nodes with many children will require many pebbles. Example 3 shows why it is a good idea to process hard subproofs first.

Example 3. Figure 3 shows a simple proof ψ with two subproofs ψ_1 ("easy") and ψ_2 ("hard"). As shown in the leftmost diagram, assume $s(\psi_1, <_T^1) = 4$ and $s(\psi_1, <_T^2) = 5$. 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 $\rho(\psi)$ can be pebbled. Therefore, $s(\psi, <_T) = s(\psi_j, <_T^j) + 1$ where $<_T = <_T^i ::: <_T^j$ and i and j are the indexes of the subproofs pebbled, respectively first and last. Choosing to pebble the easy subproof first results in $s(\psi, <_T) = 6$, while pebbling the hard one first gives $s(\psi, <_T) = 5$. This is a simplified situation with two subproofs that are independent in the sense that pebbling one of them does not influence the pebble number of the other, which is not true if they share nodes.

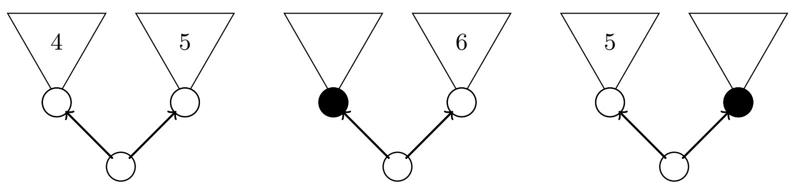

Fig. 3: Hard subproof first

The two rightmost diagrams are confusing, because they give the impression that the number of pebbles used in the subproofs increased to 6 and 5, respectively. In fact, 6 and 5 are the number of pebbles used for the whole proof. This has to be improved. Or the the diagrams could simply be removed. The text and the leftmost diagram are sufficient. Perhaps we even don't need any diagram.

7.2 Last Child Heuristic

As discussed in Section 4 in the proof of Theorem 1, the best moment to unpebble a node v is as soon as all its last child w.r.t. a topological order $<_T$ 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 $<_T$. Before the traversal, $h(v)$ is set to 0 for every node v. During the traversal $h(v)$ is increased by 1, if v is the last child of the current node w.r.t. $<_T$. For this heuristic, $H = \mathbb{N}$ and $\prec$ is the *smaller than* relation on $\mathbb{N}$. To some extent, this heuristic is paradoxical: v may be the last child of a node v' according to $<_T$, but pebbling it early may result in another topological order $<_T^*$ according to which v is not the last child of v'. Nevertheless, sometimes

This means it might make sense to iterate pebbling with last child heuristic. Does iteration lead to a fixpoint? Or can it oscillate? These are just some interesting questions beyond the scope of this paper.

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 bottommost node is the last child of the top right node in every topological order.

Any proof corresponding to this DAG has a tautological clause. Are you able to find a better example, where this doesn't happen?

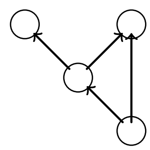

Fig. 4: Bottommost node as necessary last child of right topmost node

7.3 Node Distance Heuristic

In Example **??** 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 Heuristic prefers to pebble nodes that are close to pebbled nodes. It does this by calculating spheres with a limited radius 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:

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

where $<_P$ denotes the total order on the initial sequence of pebbled nodes P, i.e. nodes in spheres that were pebbled later are preferred. So $H = \mathbb{Z} \times \mathbb{N} \times P$ and $\prec$ is the lexicographic order using, respectively, the *smaller than* relation on $\mathbb{Z}$ and $\mathbb{N}$ and $<_P$ on P. 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 this radius, another heuristic has to be used.

7.4 Decay Heuristics

Decay Heuristics denote a family of meta heuristics. The idea is to not only use the measure of a single node, but also to include the measures of its premises. Such a heuristic has four parameters: a heuristic function $h_u : V \to H$, a recursion depth $d \in \mathbb{N}$, a decay factor $\gamma \in \mathbb{R}^+ \cup \{0\}$ and a family of combining functions $com : H^n \to H$ for $n \in \mathbb{N}$.

Table 1: Proofs benchmarks and statistics

Benchmark Category	Set 1 Number of Proofs	Set 2 Number of Proofs
QF_UF	3936	199
QF_IDL	446	184
QF_LIA	587	450
QF_UFIDL	16	2
QF_UFLIA	123	78
QF_RDL	30	1
Sum	914	5138
Total number of proof nodes	4926946	441244454
Average number of proof nodes	85879	5391

The resulting heuristic function $h : V \rightarrow H$ is defined with the help of the recusive function $rec : (V \times \mathbb{N}) \rightarrow H$:

$$
\begin{aligned}
rec(v, 0) &:= h_u(v) \\
rec(v, k) &:= h_u(v) + com(rec(p_1, k-1), \ldots, rec(p_n, k-1)) * \gamma \quad \text{where } P_v^{\psi} = \{p_1, \ldots, p_n\} \\
&\qquad \text{and } k \in \{1, \ldots, d\} \\
h(v) &:= rec(v, d)
\end{aligned}
$$

overfull hboxes have to be fixed.
When talking about TraceCheck proofs in the experimental section, we should mention the RUP format too. The paragraphs I wrote in the IJCAR paper could be reused and adapted.

8 Experiments

All the pebbling algorithms and heuristics described in the previous sections have been implemented in the functional programming language Scala[1] as part of the Skeptik library[2].

In order to evaluate the heuristics, experiments were run on two sets of test cases, consisting of proofs produced by the SMT-solver veriT[3] on unsatisfiable benchmarks from the SMT-Lib[4]. The details on the number of proofs per SMT category and their size in proof nodes are shown in Table 1. The proofs were translated into pure resolution proofs by considering every non-resolution inference as an axiom.

By default Skeptik stores proofs using a topological order that is found in a Bottom-up fashion, by visiting premises in the order they were parsed. The

[1] http://www.scala-lang.org/
[2] https://github.com/Paradoxika/Skeptik
[3] http://www.verit-solver.org/
[4] http://www.smtlib.org/

Table 2: Total compression ratios

Heuristic	(1)	(2)
Children	-1 %	0,27 %
LastChild	0,7 %	7,8 %

compression of space measures of heuristics is compared to this default topological order. Note that it is another Bottom-up heuristic, which is referred to as *uncompressed*.

The experiments were executed on the Vienna Scientific Cluster[5] VSC-2. Each algorithm was executed in a single core and had up to 16 GB of memory available. This amount of memory has been useful to compress the biggest proofs (with more than 10^6 nodes).

The overall results of the experiments are shown in Table 3 and **??**. The compression ratios are computed according to the formulas (1) and (2), in which ψ ranges over all the proofs in the corresponding set of test cases and $sp(\psi, heuristic)$ denotes the space measure of ψ w.r.t. the topological order computed by the respective heuristic.

$$\frac{\sum sp(\psi, uncompressed) - \sum sp(\psi, heuristic)}{\sum sp(\psi, uncompressed)} \tag{1}$$

$$\frac{\sum \frac{sp(\psi, uncompressed) - sp(\psi, heuristic)}{sp(\psi, uncompressed)}}{total\ number\ of\ proofs} \tag{2}$$

8.1 Test set 1

The following heuristics were evaluated using this test set:

Childen: the Bottom-Up version of the Number of Children Heuristic, see 7.1
LastChild: the Bottom-Up version of the Last Child Heuristic, see 7.2
ChildenTD: *experiments on the Top-down version of the Number of Children Heuristic are still running*
LastChildTD: *experiments on the Top-down version of the Last Child Heuristic are still running*

The results suggest, that smaller proofs benefit more from the heuristics. This claim is supported by Figure **??**, which compares the proof length in nodes to the achieved compression.

8.2 Test set 2

The following heuristics were evaluated using this test set:

[5] http://vsc.ac.at/

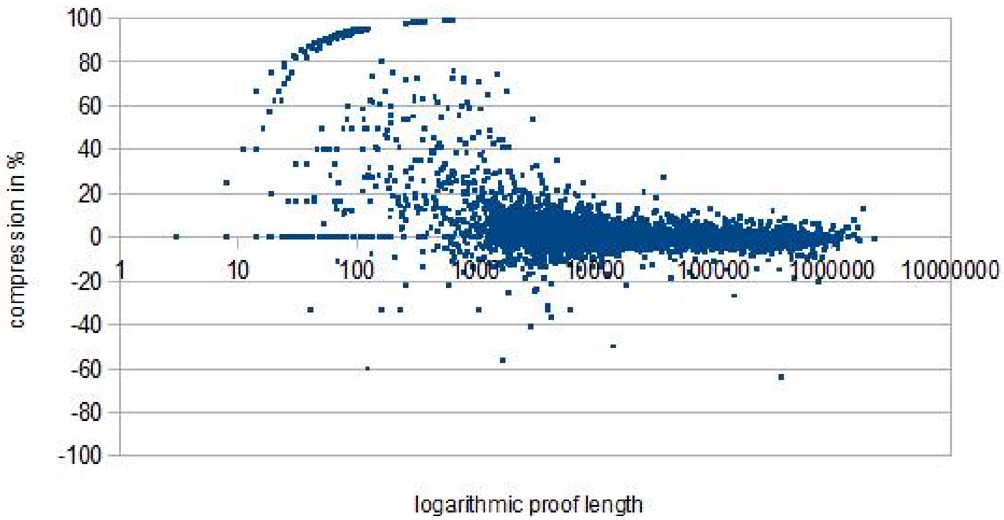

Fig. 5: Proof length compared to compression of Last Child Heuristic

Childen: the Bottom-Up version of the Number of Children Heuristic (7.1)
LastChild: the Bottom-Up version of the Last Child Heuristic (7.2)
ChildenTD: the Top-down version of the Number of Children Heuristic
LastChildTD: the Top-down version of the Last Child Heuristic
Distance3: the Bottom-Up version of the Node Distance Heuristic with a maximum radius of 3
Distance3TD: the Top-down version of the Node Distance Heuristic with a maximum radius of 3
LCllmax: the Decay Heuristic, using **LastChild** as underlying heuristic, with $\gamma = 0.5$, $d = 1$ and $com = max(\ldots)$
LChhmax: the Decay Heuristic, using **LastChild** as underlying heuristic, with $\gamma = 3$, $d = 7$ and $com = max(\ldots)$
LCllavg: the Decay Heuristic, using **LastChild** as underlying heuristic, with $\gamma = 0.5$, $d = 1$ and $com = average(\ldots)$
LChhavg: the Decay Heuristic, using **LastChild** as underlying heuristic, with $\gamma = 3$, $d = 7$ and $com = average(\ldots)$

The results of this set imply, that Bottom-up heuristics produce significantly better topological orders than their Top-down pendants. Also just like **Test Set 1** the results presented in Table 3 show that small proofs benefit more from the heuristics.

We could evaluate the effect of RPI and LUV on the space measure

We could iterate the last child heuristic a few times to see if we get better compression.

.

Table 3: Total compression ratios

Heuristic	Compression (1)	Compression (2)
Children	8,4 %	8,9 %
LastChild	25,1 %	36 %
ChildenTD	-15,8 %	0,7 %
LastChildTD	-16,7 %	5,1 %
Distance3	-0,7 %	-3,2 %
Distance3TD	-30,8 %	-37,6 %
LCllmax	25,4 %	36,6 %
LChhmax	26,6 %	37,4 %
LCllavg	25,5 %	36,7 %
LChhavg	25,7 %	34,4 %

9 Conclusions and Future Work

Bruno
Acknowledgments:

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. Chan, S.M.: Pebble games and complexity (2013)
3. 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)
4. 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
5. 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)
6. Hertel, P., Pitassi, T.: Black-white pebbling is pspace-complete. In: Electronic Colloquium on Computational Complexity (ECCC). vol. 14 (2007)
7. Kasai, T., Adachi, A., Iwata, S.: Classes of pebble games and complete problems. SIAM Journal on Computing 8(4), 574–586 (1979)
8. Nordström, J.: Narrow proofs may be spacious: Separating space and width in resolution. SIAM Journal on Computing 39(1), 59–121 (2009)
9. 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)
10. Pippenger, N.: Advances in pebbling. Springer (1982)
11. Sethi, R.: Complete register allocation problems. SIAM journal on Computing 4(3), 226–248 (1975)
12. 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