Greedy Pebbling: Towards the Compression of the Space of Proofs

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. Bruno

1 Introduction

Bruno

2 Introduction

Bruno

Propositional Resolution proofs usually are huge. So huge that when processing such proofs memory limits are reached and exceeded. However not the whole proof needs to be kept in memory. Proof nodes that are not used further on can be removed from memory. Information when to remove which node can be added as extra lines in the proof output. The maximum number of proof nodes that have to be kept in memory at once using deletion information is called the space measure. This measure is closely related to the Black Pebbling Game [**?**,**?**]. A strategy for this game corresponds to a topological ordering of the proof nodes plus deletion information. Finding an optimal strategy for the Black Pebbling Game is PSPACE-complete [**?**] and therefore not a feasible approach to compress big proofs. This paper investigates heuristic approaches to the problem.

3 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*.

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

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.*

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

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 both 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.
Let v be a node in ψ, then P_v^ψ denotes its premises and C_v^ψ its children in the corresponding graph.

Definition 2 (Topological Order). *A topological order of a proof ψ is a total order relation $<_T$ on V_ψ, such that*

$$\text{for all } v \in V_\psi, \text{ for all } p \in P_v^\psi \text{ hold } p <_T v$$

□

A topological order $<_T$ can be represented by a sequence of proof nodes $(s_1, \ldots, s_n)$, by defining $<_T := \{(s_i, s_j) \mid 1 \leq i < j \leq n\}$.
Note that there are proofs, for which exponentially many different topological orders exists:
Consider a perfect binary tree with m axioms and $n = 2m - 1$ nodes. It is easy to see that there is a proof corresponding to this graph. There is at least one topological order $(s_1, \ldots, s_n)$, which for example can be constructed using one of the algorithms described in section 5. 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. 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 exponential in m and therefore also in n.
This means that when looking for a good topological order w.r.t. any characteristic, enumeration is not a feasible option.

By loading nodes into memory, following a topological order, proofs can be traversed in a top-down fashion, for example to check the correctness or manipulate the proof.
Doing such a traversal, some nodes can (temporarily) be removed from memory. This gives rise to the following definition:

Definition 3 (Space measure). *The* space measure *of a proof ψ is the maximum number of nodes that have to be kept in memory at once, when traversing ψ in a top-down fashion, following the topological order $<_T$.* □

4 Pebbling Game

Pebbling games were first formulated in the 70's and were used to model the expressivity of programming languages [**?**,**?**] and compiler construction [**?**]. More recently, pebbling games have been used to investigate various questions in parallel complexity [**?**] and in proof complexity [**?**,**?**,**?**]. They are used to get bounds for space and time requirements and tradeoffs between the two measures [**?**].
The motivation of this work is to compress propositional resolution proofs w.r.t. the space measure. However the results can easily be transferred to more general kinds of DAGs. For our purposes we use a slight variation of the Black Pebbling Game presented in [**?**,**?**] and since we don't consider any other pebbling game in this work, we will omit the *Black*.
In the following, we use the verbs *pebble* and *unpebble* as abbreviations for *put a pebble on* and *remove a pebble from*.
A node is said to be *pebbleable*, if all its predecessors are pebbled.

Definition 4 (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. Pebbles are put on nodes according to the following rules:*

1. *If all predecessors of a node v are pebbled, then v can be pebbled.*
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 is pebbling 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 [**?**,**?**] introduces another rule with 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 [**?**].
Omitting rule 3 allows for pebbling strategies with lower pebbling numbers ([**?**] has an example on page 1). However [**?**] shows that this possibly has the cost of needing exponentially more moves in the game.
In [**?**] it is shown, that deciding the question, weather the pebbling number of a graph G and node s is smaller than k, is PSPACE-complete. The same question has shown to be NP-complete [**?**] when rule 3 is included.
In the version of the game we use and the aim to construct a strategy with a low pebbling number, unpebbling moves can be dropped from the pebbling strategy, because they are given implicitly. A node is unpebbled right after all its successors have been pebbled. It can't be unpebbled earlier, since otherwise its yet unpebbled successors can't be pebbled. Unpebbling it later could increase the pebbling number.
In the following sections, when constructing a pebbling strategy for a proof ψ, the distinguished node s, will always be the root node Γ_ψ. Instead of predecessors and successors, we will speak of premises and children. A pebbling strategy with implicit unpebbling moves directly corresponds to a topological order, defined in 2.

4.1 SAT encoding

To find the pebbling number of a proof, the question weather this proof can be pebbled using no more than k pebbles, can be translated into a propositional satisfiability problem.
Let ψ be a proof with nodes $v_1, \ldots, v_n$ and let $v_n = \Gamma_\psi$. It is assumed that the strategy contains of exactly n pebbling moves. This assumption directly reflects rule 3 of the pebbling game, defined above.

Variables 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 true that pebble x is placed on node v_j at move t.

Constraints The following constraints, combined conjunctively, are satisfiable *iff* there is a pebbling strategy for ψ, using at most k pebbles.
The sets A_ψ and P_j^ψ are interpreted as sets of indices of the respective nodes.

1. Root is pebbled
$$\bigvee_{x=1}^{k} p_{x,n,n}$$
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 at each move

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

5. For pebbling a node, its premises have to be pebbled and only 1 node is pebbled each move

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

This encoding is polynomial, both in n and k. However constraint 5 accounts to $O(n^3 * k^2)$ clauses. Many propositional resolution proofs have more than 1000 nodes and the pebbling number should be bigger than 100, which adds up to 10^{13} clauses for constraint 5 alone. This unfortunately are too many, so this encoding does not show results for reasonably big examples.

Proof?

5 Greedy Pebbling Algorithms

In this section we present two different approaches for constructing pebbling strategies using heuristics. They are called Top-down and Bottom-up Pebbling. Their names reflect the traversal direction in which the algorithms operate on proofs.

5.1 Top-down Pebbling

Algorithm 1 displays Top-down Pebbling, which constructs a topological order of a proof ψ by traversing it from one axiom to its root node Γ_ψ. 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 form of a set P, initialized as A_ψ. When pebbling a node v, it is removed from P and added to the sequence representing the topological order. For all its children it is checked, weather they are now available for pebbling and if so they are added to P.
Since ψ is a DAG, P being empty implies that all nodes have been pebbled 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 is 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**
            $P = P \cup \{c\}$;
    $P = P \setminus \{v\}$;
    $S = S \oplus v$ //$\oplus$ is the concatenation of sequences;
return S;

Algorithm 1: `Top-down Pebbling`

5.2 Bottom-up Pebbling

Algorithm 2 and 3 display Bottom-up Pebbling, which constructs a topological order of a proof ψ by traversing it from its root node Γ_ψ to one of its axioms. The algorithm constructs the order by visiting nodes and their premises recursively. At every node it is decided heuristically in what order the premises are visited. After visiting its premises, the node itself is added to the current sequence of nodes. Since axioms don't 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 recursive premises of the root, after the root has been visited a topological order is found. As opposed to Top-down Pebbling, this is more of a structural approach, instead of actually playing the pebbling game.

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

S is the empty sequence;
$V = \emptyset$;
return `visit(`ψ,Γ_ψ,V,S`)`;

Algorithm 2: `Bottom-up Pebbling`

5.3 Top-down vs Bottom-up Pebbling

In principle every topological order of a given proof can be constructed using Top-down or Bottom-up Pebbling. Both algorithms have linear run-time in the proof size (given that the heuristic choice only requires constant time). The question relevant for this paper is, which approach behaves better for constructing orders, that produce small space measures. It turns out Bottom-up Pebbling wins the

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
1 $V_1 = V \cup \{v\}$;
2 $D = P_v^\psi \setminus V$;
3 $S_1 = S$
4 **while** D *is not empty* **do**
5 choose $p \in D$ heuristically;
6 $D = D \setminus v$;
7 $S_1 = S_1 \oplus visit(\psi, v, V, S)$ //$\oplus$ is the concatenation of sequences;
8 **return** $S_1 \oplus v$;

Algorithm 3: `visit`

race.
The reason is that Top-down Pebbling often encounters situations, where it pebbles nodes that are far away from the previously pebbled nodes. Far away here is meant with respect to the amount of undirected edges between nodes in the graph. Example 1 displays this issue.
Bottom-up Pebbling does not suffer from this unlocality issue that much, because queuing up the processing of premises enforces local pebbling, which is shown in example 2.

Example 1. Consider the graph shown in figure 3 and the initial sequence of nodes $(1, 2, 3)$, which was found by some heuristic. 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 pebbled next. The subsequent pebbling moves are reasonable and can be found by a greedy heuristic. 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.
Pebbling '7' after '3' can be done with a heuristic that measures the number of edges between pebbled and pebbleable nodes. However for every d, one can easily construct a similar example in which '7' and '3' have $d + 1$ edges between them. Computing d-Spheres w.r.t. this measure can be exponential in d. Therefore only relatively small spheres can be used, in order not to slow down the whole process too much.

Example 2. Figure 2 shows a possible computation of Bottom-Up Pebbling on the same graph as presented in Figure 3. The hatched nodes were chosen by the heuristic to be processed before the respective other premise. Similar to the Top-down Pebbling scenario, this results in the initial sequence $(1, 2, 3)$. However the choice where to go next is predefined by the hatched nodes. Since one of its

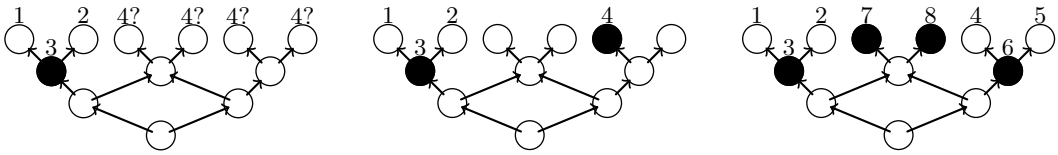

Fig. 1: Top-down Pebbling issue

premises (3) is completely processed, the other premise is visited next. The result is that node '7' can be pebbled early and at no point more than 3 pebbles will be used for pebbling the root node.
Note that this result is independent of the heuristic choices.

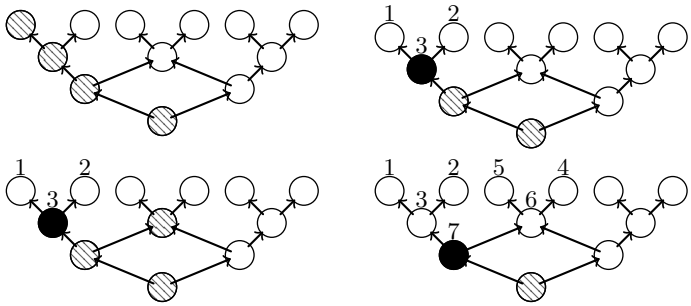

Fig. 2: Example 1 with Bottom-Up Pebbling

6 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. Top-down-, as well as Bottom-Up-, Pebbling select one 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.

6.1 Number of Children Heuristic

This heuristic uses the number of children nodes of a node v the deciding characteristic, i.e. $h(v) = |C_v^\psi|$, $H = \mathbb{N}$ and $\prec$ is the natural smaller relation.
The motivation of 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 with two subproofs. The numbers in the subproofs denote the pebbling number, after it has been pebbled. The left, easy subproof needs 4 pebbles and the right, hard subproof requires 5 pebbles. After pebbling one of them, the pebble on its root node has to be kept there until the other is pebbled fully. This increases the pebbling number of the other subproof by one. So pebbling the easy subproof first increases the overall pebbling number to 6, while pebbling the hard one first leaves it at 5.
Note that this is a simplified situation with two independent subproofs, for which pebbling one does not influence the pebbling number of the other, which is not true if they share nodes.

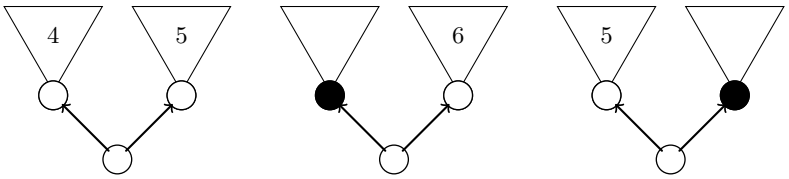

Fig. 3: Hard subproof first

6.2 Last Child Heuristic

In section 4 we explained the implicit unpebbling of a node v as soon as all of its children have been pebbled. More precisely, v can be unpebbled as soon as the its last child, w.r.t. a topological order $<_T$, is pebbled. The idea of this heuristic is to prefer nodes that are last children of other nodes.
Pebbling a node, which allows another one to be unpebbled, is always a good move. It does not increase the pebbling number, it might decrease the current number of pebbles used, if more than one premise can be unpebbled, and it possibly opens up new nodes for pebbling.
For determining the number of premises of which a node is the last children of, it first has to be traversed top-down 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. $<_T$. So for this heuristic, just like as the Number of Children Heuristic, $H = \mathbb{N}$ and $\prec$ is the natural smaller relation.
To some extent, this heuristic is a paradox, because pebbling the last child v of some node early, might results in v not being the last child anymore.
However some nodes are forced to be the last child of another node with more than one child by the structure of the proof, as shown in figure 4, where the bottom node is the last child of the top right node in every topological order.

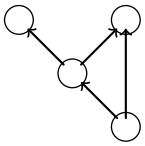

Fig. 4: Forced last child

6.3 Node Distance Heuristic

In section Section 5.3 the issue with non local pebbling was explained. The Node Distance Heuristic searches for pebbled nodes that are close to the decision node. It does this by calculating spheres with a limited radius around nodes. A sphere with radius r around the node v in the graph $G = (V, E)$ is defined in the following way:

$$K_r^G(v) =: \{p \in V \mid \text{ there are at most } r \text{ (undirected) edges between } p \text{ and } v\}$$

Using these spheres, the characteristic for this heuristic is defined as follows:

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

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 the order used is the lexicographic order of two times the natural smaller relation and $<_P$. The spheres $K_r(v)$ can grow exponentially in r. Therefore the maximum sphere calculated has to be limited and if no pebbled node is found in any of the calculated spheres, another heuristic has to be used.

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

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

Table 1: Number of proofs per benchmark category

Benchmark Category	Number of Proofs
QF_UF	3907
QF_IDL	475
QF_LIA	385
QF_UFIDL	156
QF_UFLIA	106
QF_RDL	30

7 Experiments

`LowerUnivalents` and `LUnivRPI` have been implemented in the functional programming language Scala[1] as part of the Skeptik library[2]. `LowerUnivalents` has been implemented as a recursive `delete` improvement.

The algorithms have been applied to 5 059 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 are shown in Table 1. The proofs were translated into pure resolution proofs by considering every non-resolution inference as an axiom.

The experiment compared the following algorithms:

LU: the `LowerUnits` algorithm from [**?**];
LUniv: the `LowerUnivalents` algorithm;
RPILU: a non-sequential combination of `RPI` after `LowerUnits`;
RPILUniv: a non-sequential combination of `RPI` after `LowerUnivalents`;
LU.RPI: the sequential composition of `LowerUnits` after `RPI`;
LUnivRPI: the non-sequential combination of `LowerUnivalents` after `RPI` as described in Sect. **??**;
RPI: the `RecyclePivotsWithIntersection` from [**?**];
Split: Cotton's `Split` algorithm ([**?**]);
RedRec: the `Reduce&Reconstruct` algorithm from [**?**];
Best RPILU/LU.RPI: which performs both `RPILU` and `LU.RPI` and chooses the smallest resulting compressed proof;
Best RPILU/LUnivRPI: which performs `RPILU` and `LUnivRPI` and chooses the smallest resulting compressed proof.

For each of these algorithms, the time needed to compress the proof along with the number of nodes and the number of axioms (i.e. *unsat core* size) have been measured. Raw data of the experiment can be downloaded from the web[5].

[1] http://www.scala-lang.org/
[2] https://github.com/Paradoxika/Skeptik
[3] http://www.verit-solver.org/
[4] http://www.smtlib.org/
[5] http://www.matabio.net/skeptik/LUniv/experiments/

Table 2: Total compression ratios

Algorithm	Compression	Unsat Core Compression	Speed
LU	7.5 %	0.0 %	22.4 n/ms
LUniv	8.0 %	0.8 %	20.4 n/ms
RPILU	22.0 %	3.6 %	7.4 n/ms
RPILUniv	22.1 %	3.6 %	6.5 n/ms
LU.RPI	21.7 %	3.1 %	15.1 n/ms
LUnivRPI	22.0 %	3.6 %	17.8 n/ms
RPI	17.8 %	3.1 %	31.3 n/ms
Split	21.0 %	0.8 %	2.9 n/ms
RedRec	26.4 %	0.4 %	2.9 n/ms
Best RPILU/LU.RPI	22.0 %	3.7 %	5.0 n/ms
Best RPILU/LUnivRPI	22.2 %	3.7 %	5.2 n/ms

The experiments were executed on the Vienna Scientific Cluster[6] 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 2. The compression ratios in the second column are computed according to formula (1), in which ψ ranges over all the proofs in the benchmark and ψ' ranges over the corresponding compressed proofs.

$$1 - \frac{\sum |V_{\psi'}|}{\sum |V_{\psi}|} \tag{1}$$

The unsat core compression ratios are computed in the same way, but using the number of axioms instead of the number of nodes. The speeds on the fourth column are computed according to formula (2) in which d_ψ is the duration in milliseconds of ψ's compression by a given algorithm.

$$\frac{\sum |V_{\psi}|}{\sum d_{\psi}} \tag{2}$$

For the `Split` and `RedRec` algorithms, which must be repeated, a timeout has been fixed so that the speed is about 3 nodes per millisecond.

Figure **??** shows the comparison of `LowerUnits` with `LowerUnivalents`. Subfigures (a) and (b) are scatter plots where each dot represents a single benchmark proof. Subfigure (c) is a histogram showing, in the vertical axis, the proportion of proofs having *(normalized) compression ratio difference* within the intervals showed in the horizontal axis. This difference is computed using formula (3) with v_{LU} and v_{LUniv} being the compression ratios obtained respectively

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

by `LowerUnits` and `LowerUnivalents`.

$$\frac{v_{\mathrm{LU}} - v_{\mathrm{LUniv}}}{\frac{v_{\mathrm{LU}}+v_{\mathrm{LUniv}}}{2}} \tag{3}$$

The number of proofs for which $v_{\mathrm{LU}} = v_{\mathrm{LUniv}}$ is not displayed in the histogram. The *(normalized) duration differences* in subfigure (d) are computed using the same formula (3) but with v_{LU} and v_{LUniv} being the time taken to compress the proof by `LowerUnits` and `LowerUnivalents` respectively.

As expected, `LowerUnivalents` always compresses more than `LowerUnits` (subfigure (a)) at the expense of a longer computation (subfigure (d)). And even if the compression gain is low on average (as noticeable in Table 2), subfigure (a) shows that `LowerUnivalents` compresses some proofs significantly more than `LowerUnits`.

It has to be noticed that veriT already does its best to produce compact proofs. In particular, a forward subsumption algorithm is applied, which results in proofs not having two different subproofs with the same conclusion. This results in `LowerUnits` being unable to reduce unsat core. But as `LowerUnivalents` lowers non-unit subproofs and performs some partial regularization, it achieves some unsat core reduction, as noticeable in subfigure (b).

The comparison of the sequential `LU.RPI` with the non-sequential `LUnivRPI` shown in Fig. **??** outlines the ability of `LowerUnivalents` to be efficiently combined with other algorithms. Not only compression ratios are improved but `LUnivRPI` is faster than the sequential composition for more than 80 % of the proofs.

8 Conclusions and Future Work

Bruno

Acknowledgments: