Improved Single Pass Algorithms for Resolution Proof Reduction ⋆

Ashutosh Gupta

IST, Austria

Abstract. Unsatisfiability proofs find many applications in verification. Today, many SAT solvers are capable of producing *resolution proofs* of unsatisfiability. For efficiency smaller proofs are preferred over bigger ones. The solvers apply proof reduction methods to remove redundant parts of the proofs while and after generating the proofs. One method of reducing resolution proofs is *redundant resolution reduction*, i.e., removing repeated pivots in the paths of resolution proofs (*aka* Pivot recycle). The known single pass algorithm only tries to remove redundancies in the parts of the proof that are trees. In this paper, we present *three* modifications to improve the algorithm such that the redundancies can be found in the parts of the proofs that are DAGs. The first modified algorithm covers greater number of redundancies as compared to the known algorithm without incurring any additional cost. The second modified algorithm covers even greater number of the redundancies but it may have longer run times. Our third modified algorithm is parametrized and can trade off between run times and the coverage of the redundancies. We have implemented our algorithms in OpenSMT and applied them on unsatisfiability proofs of 198 examples from plain MUS track of SAT11 competition. The first and second algorithm additionally remove 0.89% and 10.57% of clauses respectively as compared to the original algorithm. For certain value of the parameter, the third algorithm removes almost as many clauses as the second algorithm but is significantly faster.

1 Introduction

An unsatisfiability proof is a series of applications of proof rules on an input formula to deduce *false*. Unsatisfiability proofs for a Boolean formula can find many applications in verification. For instance, one application is automatic learning of abstractions for unbounded model checking by analyzing proofs of program safety for bounded steps [14, 13, 10]. We can also learn unsatisfiable cores from unsatisfiability proofs, which are useful in locating errors in inconsistent specifications [22]. These proofs can be used by higher order theorem provers as sub-proofs of another proof [4].

One of the most widely used proof rules for Boolean formulas is the resolution rule, i.e., if $a \vee b$ and $\neg a \vee c$ holds then we can deduce $b \vee c$. In the application of the

⋆ This work was supported by the ERC Advanced Investigator grant on Quantitative Reactive Modeling (QUAREM).

rule, a is known as *pivot*. A *resolution proof* is generated by applying resolution rule on the clauses of an unsatisfiable Boolean formula to deduce *false*. Modern SAT solvers (Boolean satisfiability checkers) implement some variation of DPLL that is enhanced with conflict driven clause learning [20, 19]. Without incurring large additional cost on the solvers, we can generate a resolution proof from a run of the solvers on an unsatisfiable formula [23].

Due to the nature of the algorithms employed by SAT solvers, a generated resolution proof may contain redundant parts and a strictly smaller resolution proof can be obtained. Applications of the resolution proofs are sensitive to the proof size. Since minimizing resolution proofs is a hard problem [17], there has been significant interest in finding low complexity algorithms that partially minimize the resolution proofs generated by SAT solvers.

In [3], two low complexity algorithms for optimizing the proofs are presented. Our work is focused on one of the two, namely Recycle-Pivots. Lets consider a resolution step that produces a clause using some pivot p. The resolution step is called *redundant* if each deduction sequence from the clause to *false* contains a resolution step with the pivot p. A redundant resolution can easily be removed by local modifications in the proof structure. After removing the redundant resolution step, a strictly smaller proof is obtained. Detecting and removing all such redundancies is hard. Recycle-Pivots is a single pass algorithm that partially removes redundant resolutions. From each clause, the algorithm starts from the clause and follows the deduction sequences to find equal pivots. The algorithm stops looking for equal pivots if it reaches to a clause that is used to deduce more than one clause.

In this paper, we present *three algorithms* that are improved versions of Recycle-Pivots. For the first algorithm, we observe that each literal from a clause must appear as a pivot somewhere in all the deduction sequences from the clause to *false*. Therefore, we can extend search of equal pivots among the literals from the stopping clause without incurring additional cost. For the second algorithm, we observe that the condition for the redundant resolutions can be defined recursively over the resolution proof structure. This observation leads to a single pass algorithm that covers even more redundancies but it requires an expensive operation at each clause in a proof. Note that the second algorithm does not remove all such redundancies because the removal of a redundancy may lead to exposure of more. Our third algorithm is parametrized. This algorithm applies the expensive second algorithm only for the clauses that are used to derive a number of clauses smaller than the parameter. The other clauses are handled as in the first algorithm. The parametrization reduces run time for the third algorithm but also reduces the coverage of the redundancy detection.

We have implemented our algorithms in OpenSMT [5] and applied them on unsatisfiable proofs of 198 examples from plain MUS track of SAT11 competition. The original algorithm removes 11.97% of clauses in the proofs of the examples. The first and the second algorithm remove 12.86% and 22.54% of the clauses respectively. The third algorithm removes almost as many clauses as the second

algorithm in lesser time for the parameter value as low as 10. We also observe similar pattern in reduction of the unsatisfiable cores of the examples.

Related work: Two kinds of methods have been proposed in the literature for the resolution proof minimization. The first kind of methods interact with the SAT solver for proof reduction. In [23], a reduced proof is obtained by iteratively calling a SAT solver on the unsatisfiable core in the proof obtained from the last iteration. These iterations run until a fixed point is reached. Subsequently, many methods were developed to obtain minimal/minimum satisfiability core with the help of a SAT solver [12, 16, 11, 6, 15, 8, 9]. Their objectives were not necessarily to obtain a smaller proof but very often a consequence of a smaller unsatisfiable core is a smaller proof. The second kind of methods operate independently from the SAT solver and post-process the resolution proofs. The methods in [21, 2] analyzes conflict graphs in the SAT solver (without re-running the solver) to find shared proof structures and attempts to obtain shared sub-proofs among the resolution proofs of the learned clauses. In [1], a method is presented that minimizes a resolution proof by heuristically reordering resolution steps using 'linking graph' between literals. In [18], the resolution proof rewriting rules [7] are iteratively applied to reorder resolution steps locally ([1] does it in a global context) and it is expected to expose some redundancies, which are removed by applying Recycle-Pivots after each iteration. This paper only aims to find algorithms that significantly minimize the proofs within low cost. Indeed, many of the above methods achieve more minimization as compare to our algorithms but with higher costs.

This paper is organized as follows. In section 2, we present our notation and the earlier known algorithm. In sections 3, 4, and 5, we present our three algorithms. In section 6, we discuss their complexities. We present our experimental results in section 7 and conclude in section 8. In appendix A, we present the proof of correctness of our algorithms.

2 Preliminaries

In this section, we will present our notation and one of the proof reduction algorithms presented in [3].

Conjunctive normal form(CNF): In the following, we will use $a, b, c, \dots$ to denote Boolean variables. A *literal* is a Boolean variable or its negation. We will use $p, q, r, s...$ to denote literals. Let s be a literal. If $s = \neg a$ then let $\neg s = a$. Let $var(s)$ be the Boolean variable in s. A *clause* is a set of literals. A clause is interpreted as disjunction of its literals. Naturally, empty clause denotes *false*. We will use $A, B, C, ...$ to denote clauses. Let C and D be clauses. We assume for each Boolean variable b, $\{b, \neg b\} \not\subseteq C$. Let $C \vee D$ denote union of the clauses, and let $s \vee C$ denote $\{s\} \vee C$. A *CNF formula* is a set of clauses. A CNF formula is interpreted as conjunction of its clauses. We will use $P, Q, R, ...$ to denote CNF formulas. Let P be a CNF formula. Let $Atoms(P)$ be the set of Boolean

variables that appear in P. Let $Lit(P) = \{a, \neg a | a \in Atoms(P)\}$. P is *satisfiable* if there exist a map $f : Atoms(P) \to \{0, 1\}$ such that for each clause $C \in P$, there is $s \in C$ for which if $s = a$ then $f(a) = 1$ and if $s = \neg a$ then $f(a) = 0$. P is *unsatisfiable* if no such map exists.

Resolution proof: A resolution proof is obtained by applying the resolution rule to an unsatisfiable CNF formula. The resolution rule states that clauses $a \vee C$ and $\neg a \vee D$ imply clause $C \vee D$. $a \vee C$ and $\neg a \vee D$ are the *antecedent* clauses. $C \vee D$ is the *deduced* clause and a is the *pivot*. Let $C \vee D = Res(a \vee C, \neg a \vee D, a)$ if for each Boolean variable b, $\{b, \neg b\} \nsubseteq C \vee D$. We say a is the *resolving literal* between the clauses $a \vee C$ and $C \vee D$. Symmetrically, $\neg a$ is the resolving literal between $\neg a \vee D$ and $C \vee D$. Resolution is known to be sound and complete proof system for CNF formulas. In particular, a CNF formula is unsatisfiable if and only if we can deduce empty clause by applying a series of resolutions on the clauses of the the formula. The following is a definition of a labelled DAG that records the series of applications of the resolution rule.

Definition 1 (Resolution proof). *A resolution proof $\mathcal{P}$ is a labeled DAG (V, L, R, cl, piv, v_0), where V is a set of nodes, L and R are maps from nodes to their parent nodes, cl is a map from nodes to clauses, piv is a map from nodes to pivot variables, and $v_0 \in V$ is the sink node. $\mathcal{P}$ satisfies the following conditions:*

(1) V is divided into leaf and internal nodes.
(2) A leaf node *v has no parents, i.e., $L(v) = R(v) = \bot$ and $piv(v) = \bot$.*
(3) An internal node *v has exactly a pair of parents $L(v)$ and $R(v)$ such that $cl(v) = Res(cl(L(v)), cl(R(v)), piv(v))$.*
(4) v_0 is not a parent of any other node and $cl(v_0) = \emptyset$.

$\mathcal{P}$ is *derived* from unsatisfiable CNF formula P if for each leaf $v \in V$, $cl(v) \in P$. Let $Lit(\mathcal{P}) = Lit(\{cl(v) | v \in V\})$. Let $children(v) = \{v' \in V | v = L(v') \vee v = R(v')\}$. If $v' \in children(v)$ then let $rlit(v, v')$ be the resolving literal between v and v', i.e., if $v = L(v')$ then $rlit(v, v') = piv(v')$ else $rlit(v, v') = \neg piv(v')$.

Since we will be dealing with the algorithms that modify resolution proofs, we may refer to a resolution proof that satisfies all the conditions except the third. We will call such an object as *proof DAG*.

Proof reduction: The resolution proofs obtained from SAT solvers may have redundant parts, which can be partially removed during the post-processing using low complexity algorithms. We focus on such an algorithm introduced in [3], namely Recycle-Pivots. The observation behind the algorithm is that if there is a node $v \in V$ such that each path from v to v_0 contains a node v' such that $piv(v) = piv(v')$ then the resolution at node v is redundant. v can be removed using an inexpensive transformation of the resolution proof. The transformed resolution proof is a strictly smaller than the original resolution proof. We will call this minimization *redundant pivot reduction*.

In figure 1, we present an algorithm RmRedundancies, which is a reproduction of Recycle-Pivots from [3]. RmRedundancies takes a resolution

global variables
(V, L, R, cl, piv, v_0) : resolution proof *visited* : $V \to \mathbb{B} = \lambda x.false$

```
fun RmRedundancies(P)
begin
  (V, L, R, cl, piv, v0) := P
  RmPivots(v0, ∅)
  visited := λx.false
  v0 := RestoreResTree(v0)
end
```

```
fun RmPivots(v: node, D: literals)
begin
1  if visited(v) then return
2  visited(v) := true
3  if piv(v) = ⊥ then return
4  if |children(v)| > 1 then D := ∅
5  if piv(v) ∈ D then
6    R(v) := ⊥
7    RmPivots(L(v), D)
8  elsif ¬piv(v) ∈ D then
9    L(v) := ⊥
10   RmPivots(R(v), D)
11 else
12   RmPivots(L(v), D ∪ {piv(v)})
13   RmPivots(R(v), D ∪ {¬piv(v)})
end
```

```
fun RestoreResTree(v: node)
begin
1  if visited(v) then return v
2  visited(v) := true
3  if piv(v) = ⊥ then return v
4  if L(v) = ⊥ then
5    v' := RestoreResTree(R(v))
6  elsif R(v) = ⊥ then
7    v' := RestoreResTree(L(v))
8  else
9    v_l := RestoreResTree(L(v))
10   v_r := RestoreResTree(R(v))
11   match (piv(v) ∈ cl(v_l), ¬piv(v) ∈ cl(v_r)) with
12     | (true, true) -> v' := v
13     | (true, false) -> v' := v_r
14     | (false, _) -> v' := v_l
15 if v = v' then
16   cl(v) := Res(v_l, v_r, piv(v))
17 else
18   for each u : v = L(u) do L(u) := v' done
19   for each u : v = R(u) do R(u) := v' done
20 return v'
end
```

Fig. 1. RmRedundancies, a dual pass resolution proof reduction algorithm from [3].

proof $\mathcal{P}$ as input and only removes the redundancies in parts of $\mathcal{P}$ that are trees. This algorithm traverses $\mathcal{P}$ twice using two algorithms, namely RmPivots and RestoreResTree. RmPivots detects and flags the redundant clauses in tree like parts of resolution. RestoreResTree traverses the flagged resolution proof and removes the redundant clauses using appropriate transformations.

RmPivots recursively traverses the proof DAG in depth first manner. RmPivots takes a node v as the first input argument. At line 1-2 using map *visited*, it ensures that v is visited only once. At line 3, if v is a leaf then the algorithm returns. The algorithm also takes a set of literals D as the second input argument. D is a subset of the resolving literals that have appeared along the path using which DFS has reached v via the recursive calls at lines 7, 10, 12, and 13. At line 4, D is assigned empty set if v has multiple children. Consequently, D contains only the resolving literals that appeared after the last node with multiple children was visited. At line 5 and 8, if $piv(v)$ or $\neg piv(v)$ is found in D, then we have detected a redundant resolution step. The algorithm flags the detected redundant clause by removing one of the parent relations at lines 6 or 9. This modification in parent relations violates the conditions of a resolution proof.

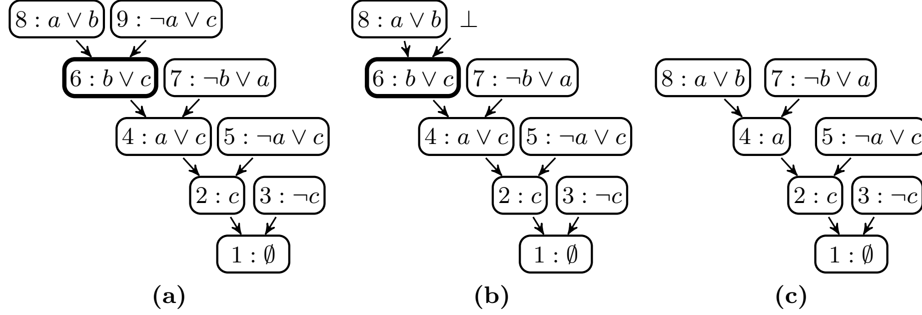

Fig. 2. Each node is labelled with a clause and assigned a number as a node id. The left parent corresponds to L parent and the right parent corresponds to R parent. The pivot used for producing a node can be inferred by looking at the clauses of the node and its parents. (a) An example resolution proof with a redundant resolution at node 6. (b) A proof DAG obtained after running RmPivots. (c) A resolution proof obtained after running RmRedundancies.

After running RmPivots, RmRedundancies calls RestoreResTree to remove the flagged clauses. RestoreResTree traverses the proof DAG in the order of parents first. RestoreResTree takes a node v as input and returns a node v' that has a valid sub-proof (line 15–16) and replaces v in the resolution proof (line 17–20). If v is a flagged node then v' is the node that also replaces the remaining parent of v (line 4–6). If v was originally not flagged then it may happen that one of its new parents v_l and v_r may not contain the literals corresponding to $piv(v)$ (line 9–12). In this case, v is treated as a flagged node(line 13–14). Please look in [3] for more detailed description of RestoreResTree.

Example 1. Consider the node 6 of the resolution proof presented in figure 2(a). The resolution at node 6 is redundant because the path from 6 to sink node 1 contains node 2 and both nodes 6 and 2 have pivot a. RmPivots will reach to node 6 with $D = \{a, b, c\}$. Therefore, $R(6)$ will be assigned $\perp$. In figure 2(b), we show a proof DAG obtained after running RmPivots. The subsequent run of RestoreResTree produces a resolution proof shown in figure 2(c).

For efficiency, RmPivots not so eagerly flags clauses for removal. In the following three sections, we will present three new algorithms to replace RmPivots that will detect more redundancies without adding much additional cost. RestoreResTree is general enough such that in all the three following algorithms it will be able to subsequently restore the resolution proof.

3 Using literals of clauses for redundancy detection

In this section, we will present our first modification in RmPivots that leads to detection of more redundant resolutions without additional cost.

For each node $v \in V$, we observe that each literal in $cl(v)$ has to act as a resolving literal in some resolution step along each path to the sink v_0 because $cl(v_0)$ is empty and a literal is removed in the descendants only by some resolution (Lemma 1 in appendix A). Now consider a run of RmPivots that reaches to a node v that has multiple children. At this point of execution, RmPivots resets parameter D to empty set. Due to the above observation, we are not required to fully reset D and can safely reset D to $cl(v)$.

In figure 3, we present our first algorithm RmPivots* which is a modified version of RmPivots. The only modification is at line 4 that changes the reset operation. This modification does not require any changes in RestoreResTree.

Example 2. Consider the resolution proof presented in figure 4(a). The resolution at node 7 is redundant but RmPivots will fail to remove it because node 5 has multiple descendants and the algorithm will not look further. In RmPivots*, the literals of $cl(5)$ are added in D therefore our modification enables it to detect the redundancy and remove it. The minimized proof is presented in figure 4(b).

```
fun RmPivots*(v: node, D: literals)
begin
1   if visited(v) then return
2   visited(v) := true
3   if piv(v) = ⊥ then return
4   if |children(v)| > 1 then D := cl(v)
5   if piv(v) ∈ D then
6     R(v) := ⊥
7     RmPivots*(L(v), D)
8   elsif ¬piv(v) ∈ D then
9     L(v) := ⊥
10    RmPivots*(R(v), D)
11  else
12    RmPivots*(L(v), D ∪ {piv(v)})
13    RmPivots*(R(v), D ∪ {¬piv(v)})
end
```

Fig. 3. RmPivots*, our first improved version of RmPivots.

4 All path redundancy detection

In this section, we present our second modification in function RmPivots that considers all paths from a node to the sink to find the redundancies. This modification leads to even greater coverage in a single pass of a resolution proof but it may lead to a longer run time.

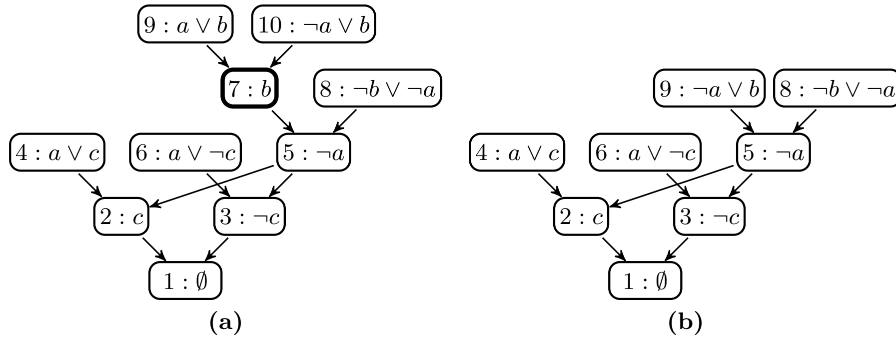

Fig. 4. (a) An example of resolution proof on which RmPivots fails to detect the redundancy at node 7. (b) Minimized form of the example.

In the second modification, we additionally compute a set of literals, which we call the expansion set, for each node to guide the removal of redundant resolutions. For a node $v \in V$, the *expansion set* $\rho(v)$ is the largest set of literals such that if some proof transformation among ancestors of v leads to appearance of literals from $\rho(v)$ in $cl(v)$ then the resolution proof remains valid. Due to the definition, $\rho(v)$ is a subset of the resolving literals that appear in all the paths from v to v_0. We cannot add all the resolving literals that appear in the paths in $\rho(v)$ because of the following reason. In a path from v to v_0, if literals s and $\neg s$ both appear as resolving literals then only the one that appears nearest to v can be added to $\rho(v)$. Otherwise, an expansion allowed by $\rho(v)$ may lead to the internal clauses that have both s and $\neg s$. If there are two paths in which s and $\neg s$ occur in opposite orders then none of them can be added to $\rho(v)$. The following equation recursively defines the expansion set for each node.

$$\rho(v) = \begin{cases} \emptyset & \text{if } v = v_0 \\ \bigcap\limits_{v' \in children(v)} \rho(v') \cup \{rlit(v,v')\} \setminus \{\neg rlit(v,v')\} & \text{otherwise.} \end{cases} \quad (1)$$

To understand the above equation, for each $v' \in children(v)$ lets assume we have the expansion set $\rho(v')$. All the paths from v to v_0 that goes via v' must have seen resolving literals $\rho(v') \cup \{rlit(v,v')\}$. If $\neg rlit(v,v')$ appears in $\rho(v')$ then we need to remove it as noted earlier. Therefore, $\rho(v)$ is the intersection of these resolving literal sets corresponding to the children of v. If $piv(v)$ or $\neg piv(v)$ is in $\rho(v)$ then $\rho(v)$ allows removal of the resolution step at v.

In figure 5, we present the second modified algorithm ALL-RMPIVOTS that implements the above computation of ρ and flags the resolution steps accordingly. The algorithm can replace RMPIVOTS without any changes in RESTOREResTree. In this presentation of the algorithm, we assume that initially all nodes of V are reachable from v_0. Initially, the global variable ρ maps v_0 to $\emptyset$ and rest of the nodes to all literals appear in $\mathcal{P}$. We initialize ρ in this way because if a node that eventually becomes unreachable from v_0 and has parents that are reachable from v_0 then we can consistently compute ρ for the parents. ALL-RMPIVOTS takes a node v as input and decides to visit the node now or not. The condition at line 1 ensures that each node is visited only if it is an internal node, only once, and if its parents are already visited. ALL-RMPIVOTS traverses $\mathcal{P}$ in the reverse topological order. During the visit of v, the loop at line 6 computes $\rho(v)$ slightly differently from the recursive equation (1). Subsequently at lines 11–12, the algorithm drops the parent relations if $\rho(v)$ contains $piv(v)$ or $\neg piv(v)$. The algorithm does not remove $\neg rlit(v,v')$ from $\rho(v')$ at line 7 as per the equation (1) because $\rho(v')$ cannot contain $\neg rlit(v,v')$. That is the case because, during the earlier visit to v', if $\rho(v')$ contained $\neg rlit(v,v')$ then the edge between v' and v must have been removed. At lines 13–14, a recursive call even for an ex-parent is made because if there were other children of the ex-parent and all of whom have been visited earlier then the ex-parent must be waiting to be visited. Due to the initialization of ρ, if $children(v)$ is empty then both the if-conditions at lines 11–12 are true and both the parent relations

global variables
$\rho : V \rightarrow$ literal set $:= (\lambda x.Lit(\mathcal{P}))[v_0 \mapsto \emptyset]$ k : integer (parameter)

```
fun ALL-RMPIVOTS(v: node)
begin
1  if visited(v) ∨ piv(v) = ⊥ ∨
2    ∃v′ ∈ children(v).¬visited(v′)
3  then
4    return
5  visited(v) := true
6  for each v′ ∈ children(v) do
7    ρ(v) := ρ(v) ∩ (ρ(v′) ∪ {rlit(v, v′)})
8  done
9  v_L := L(v)
10 v_R := R(v)
11 if piv(v) ∈ ρ(v) then R(v) := ⊥
12 if ¬piv(v) ∈ ρ(v) then L(v) := ⊥
13 ALL-RMPIVOTS(v_L)
14 ALL-RMPIVOTS(v_R)
end
```

```
fun K-RMPIVOTS(v: node)
begin
1  if visited(v) ∨ piv(v) = ⊥ ∨
2    ∃v′ ∈ children(v).¬visited(v′)
3  then
4    return
5  visited(v) := true
6  if children(v) ≥ k then
7    ρ(v) := cl(v)
8  else
9    for each v′ ∈ children(v) do
10     ρ(v) := ρ(v) ∩ (ρ(v′) ∪ {rlit(v, v′)})
11   done
12 v_L := L(v)
13 v_R := R(v)
14 if piv(v) ∈ ρ(v) then R(v) := ⊥
15 if ¬piv(v) ∈ ρ(v) then L(v) := ⊥
16 K-RMPIVOTS(v_L)
17 K-RMPIVOTS(v_R)
end
```

Fig. 5. ALL-RMPIVOTS and K-RMPIVOTS are our second and third modified algorithms respectively. Each can replace RMPIVOTS. To find redundant resolutions, ALL-RMPIVOTS considers all the paths from a node to the sink. Depending on the parameter k, K-RMPIVOTS only considers the paths that contain nodes with less than k children.

are removed. Therefore, if a node eventually becomes unreachable from the sink then the node also becomes isolated. Since removal of a redundancy may expose more redundancies, the algorithm removes the redundancies partially.

Example 3. Consider the resolution proof presented in figure 6(a). The resolution at node 10 is redundant because the pivot of node 10 is variable b and literal b appears as a resolving literal on both the paths from node 10 to node 1. RMPIVOTS* fails to detect it because node 6 has multiple descendants and $cl(6)$ does not contain b. ALL-RMPIVOTS computes the following values for map ρ.

$$\rho(2) = \{b\} \quad \rho(4) = \{d, b\} \quad \rho(5) = \{\neg d, b\} \quad \rho(6) = \{a, b\} \quad \rho(10) = \{a, b, \neg c\}$$

Since $b \in \rho(10)$, ALL-RMPIVOTS detects the redundancy. In figure 6(b), the minimized proof that is obtained after consecutive run of ALL-RMPIVOTS and RESTORERESTREE is presented.

5 Redundancy detection up to k children

In this section, we present our third algorithm that only considers a fraction of paths from a node to the sink to find the redundancies. The fraction is deter-

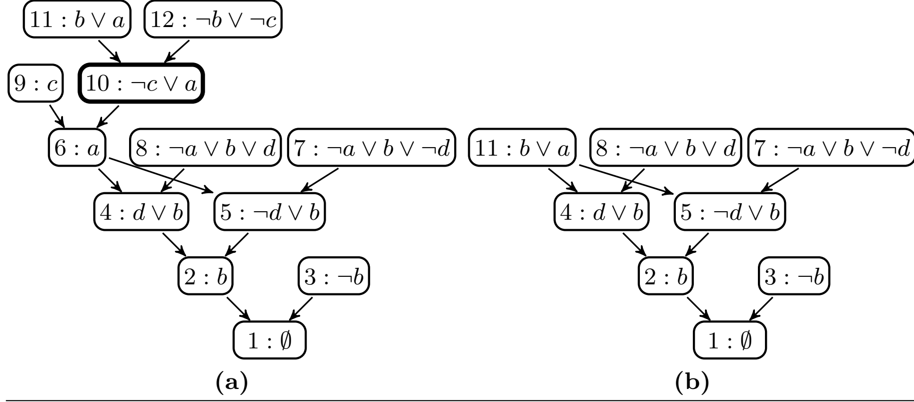

Fig. 6. (a) An example of resolution proof on which RmPivots* fails to detect the redundancy at node 10. (b) Minimized form of the example obtained by all-RMPivots.

mined by a parameter k. This algorithm only considers the paths that contain nodes with less than k children. In the following equation, we present a modified definition of the expansion set that implements the restriction.

$$\rho(v) = \begin{cases} \emptyset & \text{if } v = v_0 \\ cl(v) & \text{if } |children(v)| \geq k \\ \bigcap\limits_{v' \in children(v)} \rho(v') \cup \{rlit(v,v')\} \setminus \{\neg rlit(v,v')\} & \text{otherwise.} \end{cases}$$

If a node v has more than or equal to k children then the above equation under-approximates the expansion set by equating it to $cl(v)$. This modification may decrease the cost of computation of the expansion sets but may also lead to fewer redundancy detection.

In figure 5, we also present our third algorithm k-RMPivots using the above definition of expansion set. k-RMPivots is a modified version of all-RMPivots. At line 6, this algorithm introduces an if-condition that checks if the node v has more than or equal to k children. If the condition holds than the algorithm inexpensively computes $\rho(v)$. Otherwise, this algorithm operates as all-RMPivots.

6 Complexity

RmPivots* visits each node of the input resolution proof only once. The worst case cost of visiting each node for RmPivots* is $O(log(|Lit(\mathcal{P})|))$ because of the find and insert operations on D. Therefore, the complexity of RmPivots* is $O(|V|log(|Lit(\mathcal{P})|))$.

all-RMPivots, and k-RMPivots also visit each node of the input resolution proof only once. For each visited node, all-RMPivots iterates over children and applies the intersection operation. Since total number of edges in a

Algorithms	avg. % reduction in proof size	avg. % reduction in unsat core size	time(s)
RmPivots	11.97	0.98	1753
RmPivots*	12.86	1.10	1772
k-RMPivots with $k = 5$	19.60	2.13	2284
k-RMPivots with $k = 10$	21.14	2.41	2599
k-RMPivots with $k = 20$	21.90	2.67	2855
all-RMPivots	22.54	2.93	5000

Fig. 7. We applied our algorithms to the resolution proofs obtained by OpenSMT for 198 examples from plain MUS track of SAT11 competition. The proofs in total contain 144,981,587 nodes.

Algorithms	avg. % reduction in proof size	avg. % reduction in unsat core size	time(s)
RmPivots	6.35	1.52	17.2
RmPivots*	6.69	1.69	18.0
k-RMPivots with $k = 5$	10.10	2.76	24.1
k-RMPivots with $k = 10$	10.43	2.88	27.8
k-RMPivots with $k = 20$	10.54	2.90	31.7
all-RMPivots	10.58	2.91	51.3

Fig. 8. We also applied our algorithms to the resolution proofs obtained by OpenSMT for 132 examples from SMTLIB. The proofs in total contain 13,629,927 nodes.

resolution proof are less than twice the number of the nodes in the proof, the total number of intersection operation is linearly bounded. In worst case, each intersection may cost as much as the total number of literals in the resolution proofs. Therefore, The worst case average cost of visiting each node for all-RMPivots is $O(|Lit(\mathcal{P})|)$. k-RMPivots has worst case complexity as all-RMPivots. The complexities of all-RMPivots and k-RMPivots are $O(|V||Lit(\mathcal{P})|)$.

Since the number of literals are usually small compare to the number of nodes, we observe in experiments that although the intersections are expensive but the run times do not grow quadratically as number of nodes increases.

7 Experiments

We implemented our algorithms within an open source SMT solver OpenSMT [5]. We applied our algorithms to the resolution proofs obtained by OpenSMT for 198 unsatisfiable examples from plain MUS track of SAT11 competition. We selected an example if a proof is obtained with in 100 seconds using default options of OpenSMT and has resolution proof larger than 100 nodes.[1] These examples in total contain 144,981,587 nodes in the resolution proofs. The largest proof contains 9,489,571 nodes. In figure 7, we present the results of the experiments. We applied k-RMPivots with three values of

[1] Please find detailed results at `http://www.ist.ac.at/~agupta/sat12/index.html`

k: 5, 10, and 20. The original algorithm RmPivots removes 11.97% of nodes in the proofs of the examples. RmPivots* and all-RMPivots additionally removes 0.89% and 10.57% of nodes respectively. Even for small values of k, k-RMPivots reduces the proofs about as much as all-RMPivots, but within significantly less run times. We observe the similar pattern in reduction of the unsat cores.

The run time of RmPivots* is almost equal to RmPivots as expected. Due to the costly computations of the intersections of sets of literals, all-RMPivots shows significantly increased run time as compared to RmPivots. k-RMPivots provides a parameter that allows one to achieve the proof reduction almost as much as all-RMPivots and within the run times almost as less as RmPivots*.

We also selected 132 unsatisfiable examples from smtlib benchmarks in the theory of QF_UF to test the performance of our algorithms in another setting.[1] These examples in total produce 13,629,927 nodes in the resolution proofs. The largest proof in the examples contains 275,786 nodes. In figure 8, we present the results of applying our three algorithms to the examples. We observe the similar pattern in the results as observed in previous example set.

We note that the % of proof reduction may vary a lot for individual examples (from 0% to 48%).[1] We also observe that the two example sets have different absolute % reduction in proof sizes.

In figure 9, we plotted the relative performances of RmPivots, all-RMPivots, and k-RMPivots with $k = 10$ for the individual examples with proof sizes greater than 10000 nodes. In figure 9(a), we plot the run times of RmPivots verses the proof sizes. The dotted line in the plot denotes a linear growth. We observe that the run times grow non-linearly but for the most examples the run times are close to the linear line. In figure 9(b), we observe that the ratios of the run times of all-RMPivots and RmPivots have increasing trend with the increasing proof sizes, which follows from the difference in their complexities. In figure 9(c), we observe that the ratios of run times of k-RMPivots with $k = 10$ and RmPivots are fairly constant across the different proof sizes, which is the result of the heuristic. In figure 9(d), we observe that the ratios of the reductions in the proofs by all-RMPivots and k-RMPivots with $k = 10$ remain below 1.1 for most of the examples.

8 Conclusion

We presented three *new* single pass algorithms that can find redundant resolutions even in the resolution proofs that have DAG form, without causing significantly large run times. Since these algorithms do not try to escape a local minimum, the improvements in reductions due to these algorithms are limited. For an analogy, we can compare these algorithms with compiler optimizations in which an efficient and less compressing algorithm is always welcomed as compare to an expensive and more compressing algorithm.

These algorithms can be further harnessed by placing them in the iterative algorithm of [18]. We leave that for future work. We specially note that the addi-

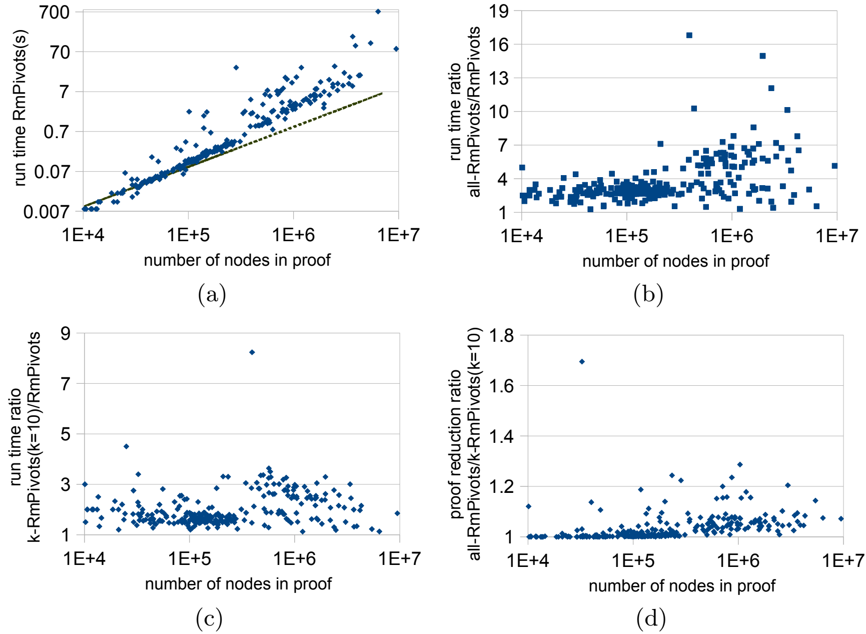

Fig. 9. (a) Run times of RmPivots vs. number of nodes in the proofs. The dotted line in the plot denotes a linear growth. (b) Ratio of the run times of all-RMPivots and RmPivots vs. number of nodes in the proofs. (c) Ratio of the run times of k-RMPivots with $k = 10$ and RmPivots vs. number of nodes in the proofs. (d) Ratio of reduction in the proofs by all-RMPivots and k-RMPivots with $k = 10$ vs. number of nodes in the proofs. For these plots, we only used the examples from the earlier benchmarks that have more than 10000 nodes.

tional rules (other than A1 and A2) for restructuring a resolution proof presented in [18] become redundant if applied in combination with our algorithms.

References

1. H. Amjad. Compressing propositional refutations. *Electr. Notes Theor. Comput. Sci.*, 185, 2007.
2. H. Amjad. Data compression for proof replay. *J. Autom. Reasoning*, 41(3-4), 2008.
3. O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Linear-time reductions of resolution proofs. In *Haifa Verification Conference*, 2008.
4. S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In J. Giesl and R. Hähnle, editors, *Automated Reasoning (IJCAR 2010)*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.
5. R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The opensmt solver. volume 6015, pages 150–153. Springer, 2010.
6. N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *SAT*, pages 36–41, 2006.

7. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *VMCAI*, 2010.
8. R. Gershman, M. Koifman, and O. Strichman. Deriving small unsatisfiable cores with dominators. In *CAV*, pages 109–122, 2006.
9. É. Grégoire, B. Mazure, and C. Piette. Local-search extraction of muses. *Constraints*, 12(3):325–344, 2007.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.
11. J. Huang. Mup: a minimal unsatisfiability prover. In *ASP-DAC*, pages 432–437, 2005.
12. I. Lynce and J. P. M. Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.
13. K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
14. K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *TACAS*, pages 2–17, 2003.
15. M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. M. Silva, and K. A. Sakallah. A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. In *SAT*, pages 467–474, 2005.
16. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *DAC*, pages 518–523, 2004.
17. C. H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *J. Comput. Syst. Sci.*, 37(1):2–13, 1988.
18. S. Rollini, R. Bruttomesso, and N. Sharygina. An efficient and flexible approach to resolution proof reduction. In *Haifa Verification Conference*, 2010.
19. J. P. M. Silva, I. Lynce, and S. Malik. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, pages 131–153. 2009.
20. J. P. M. Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
21. C. Sinz. Compressing propositional proofs by common subproof extraction. In *EUROCAST*, pages 547–555, 2007.
22. C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.
23. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiabl boolean formulas. In *SAT*, 2003.

A Proof of correctness and complexity

In this section, we will present proof of correctness of our modifications. We need to define some notation first. Let (V, L, R, cl, piv, v_0) be a proof DAG. Let ρ be defined by the recursive equation 1. Let $cl^{\rho}(v) = cl(v) \cup \rho(v)$. Let $Res^{*}(C, D, a) = (C \setminus \{a\}) \vee (D \setminus \{\neg a\})$.

In the following definition, we will define an invariant for the input of RestoreResTree. We will show that RmPivots* and all-RMPivots transforms the input resolution proof into a proof DAG that satisfies the invariant.

Definition 2 (Restorable property). *The proof DAG* (V, L, R, cl, piv, v_0) *is restorable if each internal node* $v \in V$ *satisfies the following conditions.*

(1) $L(v) \neq \bot \vee R(v) \neq \bot$
(2) *if* $L(v) = \bot \vee R(v) = \bot$ *then*

$$(3)\ cl^{\rho}(v) \supseteq \begin{cases} cl^{\rho}(L(v)) & \text{if } R(v) = \bot \\ cl^{\rho}(R(v)) & \text{if } L(v) = \bot \\ Res^{*}(cl^{\rho}(L(v)), cl^{\rho}(R(v)), piv(v)) & \text{otherwise} \end{cases}$$

Theorem 1. *If a proof DAG is restartable then* RestoreResTree *transforms the proof DAG into a resolution proof.*

We will not provide a proof for the above theorem. Please look in [3] for the proof. To prove the correctness, we will prove that during the runs of both the algorithms maintain the following invariant.

Definition 3 (Invariant). *A proof DAG* (V, L, R, cl, piv, v_0) *satisfies this invariant if each reachable from sink and internal node* $v \in V$ *satisfies the following conditions.*

(1) $L(v) \neq \bot \vee R(v) \neq \bot$
(2) *if* $R(v) \neq \bot \wedge L(v) \neq \bot$ *then* $cl(v) = Res(cl(L(v)), cl(R(v)), piv(v))$
(3) *if* $R(v) = \bot$ *then* $cl(v) \uplus \{piv(v)\} \supseteq cl(R(v))$ *and* $piv(v) \in \rho(v)$
(4) *if* $L(v) = \bot$ *then* $cl(v) \uplus \{\neg piv(v)\} \supseteq cl(L(v))$ *and* $\neg piv(v) \in \rho(v)$

where, $\uplus$ *is disjoint union.*

It can be easily checked that the invariant is a stronger property than then the restorable property. Since both RmPivots* and all-RMPivots only remove edges from a proof DAG, conditions 1 and 2 of the invariant will be true trivially. Further, first half of 3 and 4 are also true trivially. To show validity of the rest of the conditions that we need to prove the following lemma.

Lemma 1. *If the proof DAG satisfies the invariant then for each* $v \in V$, $cl(v) \subseteq \rho(v)$.

Proof. We prove it by induction over the height of the proof DAG starting from sink node. The base case at sink node is trivially true. By induction hypothesis, for a node $v \in V$, lets assume for each $v' \in children(v)$, $cl(v') \subseteq \rho(v')$. Let $v = L(v')$ and the proof for the other case is similar. Due to condition (2) and (3) of the invariant, $cl(v) \subseteq \rho(v') \cup \{piv(v')\}$. Due to definition of ρ, we can derive $cl(v) \subseteq \rho(v)$. □

Theorem 2. RmPivots* *maintains the invariant.*

Proof. The theorem is due to the previous lemma and the correctness proof of Recycle-Pivots from [3]. □

Theorem 3. all-RMPivots *maintains the invariant.*

Proof. ALL-RMPIVOTS traverses the proof DAG in reverse topological order. Therefore, a node is visited only when all the ancestors of the node has been visited. The computed value of ρ for the node will not be changed due to future changes since all the future change will not happen with in the ancestors of the node. Hence by construction, the second half of the third and fourth conditions will be satisfied. □

The above theorems are sufficient to prove the correctness of our algorithms.