Compressing Propositional Proofs by Common Subproof Extraction

Carsten Sinz

Symbolic Computation Group, WSI for Computer Science
University of Tübingen, Germany
carsten.sinz@uni-tuebingen.de

1 Introduction

Propositional logic decision procedures [1–6] lie at the heart of many applications in hard- and software verification, artificial intelligence and automatic theorem proving [7–12]. They have been used to successfully solve problems of considerable size. In many practical applications, however, it is not sufficient to obtain a yes/no answer from the decision procedure. Either a model, representing a sample solution, or a justification, why the formula possesses none is required. So, e.g. in declarative modeling or product configuration [9, 10] an inconsistent specification given by a customer corresponds to an unsatisfiable problem instance. To guide the customer in correcting his specification, a justification why it is erroneous can be of great help. In the context of model checking proofs are used, e.g., for abstraction refinement [11], or approximative image computations through interpolants [13]. In general, proofs are also important for certification through proof checking [14].

Different propositional logic solvers are already available that can generate proofs, e.g., zChaff [4], MiniSAT [6], booleforce [15], or ebddres [16]. Most of them are extensions of DPLL (Davis-Putnam-Logemann-Loveland [1, 2]) solvers, which play a dominant role in handling real-world problems. DPLL-based solvers generate *lemmas* or *learned clauses*. The derivations of these lemmas (called *proof chains*), taken together, form a resolution refutation in case the input formula is unsatisfiable. Beame *et al.* [17] have lain the theoretical groundwork for such proofs. But whereas the basic theoretical questions are resolved, practical shortcomings still exist. So the generated proofs can be of considerable size, in the order of hundreds of megabytes for realistic examples. This rises the question whether it is possible to generate smaller proofs. First steps in this direction were undertaken and include techniques like *core extraction* [14], proof generation using other algorithms than DPLL (e.g., BDD-based methods [16]), or computation of minimally unsatisfiable subsets (MUS) [18].

In this paper, we introduce a new technique for compressing proofs that is based on identifying common sub-proofs and merging them. It is intended to be used as a post-processor for proofs generated with one of the methods above. It can therefore also be combined with existing approaches for obtaining short proofs.

Merging is applied on the level of *proof chains*. Common parts of proof chains are united, thereby producing possibly shorter proofs.

2 Basic Notions

Throughout this paper, we assume propositional logic formulas in conjunctive normal form (CNF), built over a finite set of variables V. A formula F therefore is a conjunction of *clauses*, where a clause is a disjunction of *literals*, and a literal L is either a variable or its negation, i.e. $L \in V \cup \neg V$. We also use *set notation* for formulas in CNF. Assume that a formula $F = C_1 \wedge \cdots \wedge C_m = (L_{1,1} \vee \cdots \vee L_{1,k_1}) \wedge \cdots \wedge (L_{m,1} \vee \cdots \vee L_{m,k_m})$ in CNF is given. As the connectives are uniquely determined by the form, we can equivalently represent F as a set of clauses $\{C_1, \ldots, C_m\}$, where $C_i = \{L_{i,1}, \ldots, L_{i,k_i}\}$. A clause with no literals is called the *empty clause* (denoted by $\Box$), a clause with a single literal is called a *unit clause*, its sole literal is called a *unit*. By $\mathbb{B} := \{0, 1\}$ we denote the set of Boolean constants, where we identify 0 with *false* and 1 with *true*. A (partial) *valuation* (or *assignment*) is a function $\alpha : V' \rightarrow \mathbb{B}$, where $V' \subseteq V$. If $V' = V$, we say that the valuation is *complete*. A valuation can equivalently be represented as a set of non-contradictory literals, i.e. a set $A \subseteq V \cup \neg V$ with $\{x, \neg x\} \not\subseteq A$ for all variables $x \in V$. The valuation induced by such a set A is defined by $\alpha(x) = 1$ iff $x \in A$, $\alpha(x) = 0$ iff $\neg x \in A$. A valuation given as a set of literals A satisfies a formula F, if $C \cap A \neq \emptyset$ for all clauses $C \in F$. A valuation A immediately contradicts a formula F, if there is a clause $C \in F$ with $\neg L \in A$ for all literals $L \in C$. A formula F is said to be *satisfiable*, if there is a (complete) valuation that satisfies F. Otherwise it is called *unsatisfiable*.

Unit propagation is a simplifying operation on clause sets. For a formula F with unit clauses $U = \{\{u_1\}, \ldots, \{u_r\}\} \subseteq F$, unit propagation removes all clauses containing at least one $u \in U$ from F, and deletes all literals $\neg u$ (with $u \in U$) from clauses in F. We denote the clause set resulting from repeated application of unit propagation to a formula F until a fix point is reached by $\mathrm{UP}(F)$. F is satisfiable iff $\mathrm{UP}(F)$ is. Given a partial valuation A, the result of *unit propagation assuming* A, $\mathrm{UP}_A(F)$, is defined as $\mathrm{UP}_A(F) := \mathrm{UP}(F \cup A^*)$, where $A^* = \{\{a\} \mid a \in A\}$. If, after unit propagation for a clause set F and a valuation A, $\mathrm{UP}_A(F)$ contains the empty clause, assignment A contradicts F. A *conflict graph* can then be generated that captures the individual unit propagation steps that lead to the empty clause (or conflict). The nodes of the conflict graph are literals of unit clauses occurring during unit propagation plus an additional node for the conflict. We identify nodes with literals, and assume an additional "conflict literal" $\emptyset$ for the conflict node. Nodes corresponding to units from the assignment A are called *assumption nodes*, all other nodes, besides the conflict node, are called *propagation nodes*. Moreover, propagation nodes and the conflict node are marked with a clause $C \in F \cup \{\Box\}$, and $L \in C$ holds for the literal L of each node besides the conflict node. There is an edge from each literal $\neg L'$ with $L' \in C \setminus \{L\}$ to the literal L, indicating that $\{L\}$ became a new unit clause due to the presence of unit clauses $\{\neg L'\}$ for all other literals L' of C. The conflict node has either two literal nodes $L', \neg L'$ (a complementary literal pair) as predecessors, in which case it is marked with the empty clause, or it is marked with a clause $C \in F$ and has all literals $\neg L'$ for $L' \in C$ as predecessors.

Resolution [19] is a proof system for formulas in CNF, which consists of a single derivation rule which allows to infer a new clause from two given clauses C and D:

$$\frac{C \qquad\qquad D}{(C \setminus \{x\}) \cup (D \setminus \{\neg x\})}$$

A sequence of resolution steps, using clauses from F as axioms, is called a *refutation proof* for F, if the last derived clause in the sequence is the empty clause. For each unsatisfiable formula there is a resolution refutation proof witnessing that F is contradictory.

3 Propositional Proof Generation

In this section, we give a short introduction on how proofs are generated in DPLL-style solvers incorporating clause learning. We do not give the traditional presentation of the DPLL algorithm here (which can be found, e.g., in [4]), but reduce it to the facets required to understand how proofs are generated within these solvers.

The DPLL algorithm takes a propositional logic formula, say F, in CNF as input, and tries to find a satisfying assignment $\alpha : V \to \mathbb{B}$ for F (built over variable set V). It does so by extending a partial assignment—starting with the empty assignment—until a contradiction is reached. As soon as this happens, a conflict graph is generated and from this graph a *conflict clause* is derived which captures the reason, why this partial assignment is inadmissible. The conflict clause is then added to F, excluding this partial assignment for the rest of the search. Then the solver backtracks by flipping the value assigned to one of the variables in the conflict clause and continues its search, until all partial assignments have been checked. If no satisfying assignment was found, F is unsatisfiable, and a refutation proof for F may be generated.

3.1 A Proof-Theoretic View of the DPLL Algorithm

As we are only interested in proof generation, we use a slightly abstracted version of the DPLL algorithm in the following, in which the details of the backtracking-procedure are left out. This abstracted algorithm works as follows. We assume an input formula F that does not contain the empty clause (in this case the proof would be trivial):

1. Generate a minimal partial assignment π_i that does not immediately contradict F, but does so after unit propagation assuming π_i. This gives rise to a conflict graph G, which consists of the implications (unit propagations) that led to the conflict. If no such assignment π_i exists, stop (the formula then is satisfiable and no proof can be generated).
2. Add the weakest clause C_i to F, which "forbids" the partial assignment π_i, and generate a proof for C_i (called a *proof chain*) out of the conflict graph G.
3. If π_i is the empty assignment (then C_i is the empty clause $\Box$), the refutation for F is complete. Otherwise continue with Step 1.

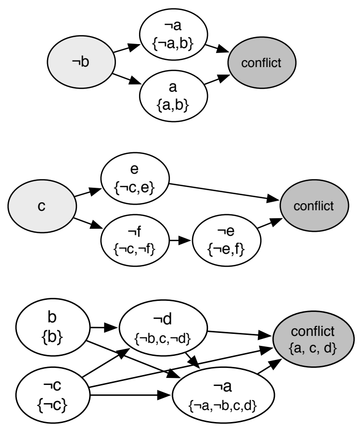
Before we illustrate the operation of this algorithm on an example, we want to comment on some details: In step 1, if no suitable assignment π_i exists, this may happen for one of the following reasons: (a) all partial assignments immediately contradict F or (b) all partial assignments that do not immediately contradict F, neither do so after unit propagation. Case (a) cannot happen, as the empty assignment never contradicts F, because $\Box \notin F$ by assumption (this invariant always holds in step 1, as the algorithm stops in step 3, as soon as the empty clause is added to F). If case (b) holds, a satisfying assignment can be constructed in the following way: start with the empty assignment and apply unit propagation. By assumption, no contradiction arises. Then add an arbitrary literal to the assignment. This does not produce an immediate conflict, as in each clause there are at least two unassigned (open) literals. Re-apply unit propagation. Again, by assumption, no conflict occurs. By repeating the process of adding literals to the assignment and applying unit propagation, one finally arrives at a complete assignment that satisfies F.

3.2 Conflict Graphs and Proof Chains

We now show in detail how proofs are constructed by the algorithm given in the last section. The construction process uses conflict graphs as intermediate objects, converts them to proof chains, and builds a complete proof out of these proof chains.

Example 1. Let $F = \{\{a, b\}, \{\neg a, b\}, \{\neg a, \neg b, c, d\}, \{\neg b, c, \neg d\}, \{\neg c, e\}, \{\neg c, \neg f\}, \{\neg e, f\}, \{a, c, d\}\}$. Assume that the algorithm first chooses the partial assignment $\pi_1 = \{\neg b\}$. Unit propagation results in the additional units a and $\neg a$, which induces a conflict. The corresponding conflict graph and the resulting proof chain are shown in Fig. 1(i). We explain below how the proof chain is constructed from the conflict graph.

(i)
(ii)
(iii)



$$\frac{\{\neg a, b\} \quad \{a, b\}}{\{b\}}$$

$$\frac{\dfrac{\{\neg e, f\} \quad \{\neg c, e\}}{\{\neg c, f\}} \quad \{\neg c, \neg f\}}{\{\neg c\}}$$

$$\frac{\{\neg c\} \quad \dfrac{\{b\} \quad \dfrac{\dfrac{\{a, c, d\} \quad \{\neg a, \neg b, c, d\}}{\{\neg b, c, d\}} \quad \{\neg b, c, \neg d\}}{\{\neg b, c\}}}{\{c\}}}{\Box}$$

Fig. 1. Conflict graphs and proof chains for Example 1. Assumption nodes are shown in light grey, conflict nodes in dark grey, and propagation nodes in white.

Now, clause $C_1 = \{b\}$ is added to F, thus excluding the partial assignment π_1 for the rest of the search. Proceeding with partial assignments $\pi_2 = \{c\}$ and $\pi_3 = \emptyset$ (shown in Fig. 1(ii) and (iii)), we obtain two further proof chains. The last one derives the empty clause. Taking all proof chains together, we have a resolution refutation proof for F.

Note, that each proof chain has a quite simple structure: each resolution step involves at least one clause from F (thus the resolution proof for a proof chain is *input* and therefore also *linear*), and each variable is resolved upon at most once (thus proof chains are also *regular*). Beame *et al.* [17] call such proofs *trivial* resolution proofs. Though the proofs for individual proof chains are trivial, this is typically not the case for the complete proof consisting of multiple proof chains.

To construct a proof chain from a conflict graph, clauses of the graph's nodes are resolved in inverse topological order: resolution starts at the conflict node and proceeds back through the graph until the sources of the graph are reached. The proof chain resulting from a given conflict graph is not unique, as there might be different topological orderings of the nodes. Our compression method makes use of this indeterminism by choosing the optimal topological ordering for merging proof chains.

4 Proof Compression Method

The input to our compression method is an already existing proof. We are using *booleforce*[1] to generate such proofs. Booleforce's output consists of a set of *proof chains*. Each proof chain consists of an *identifier* (a natural number), an *assertion* (a clause) and a *derivation* (a set of clauses, given by their identifiers). The semantics of a proof chain is such that by resolving the derivation's clauses (in a not further specified order) the chain's assertion can be proved. Assertions are represented in a DIMACS-like clause format, derivations are space-separated lists of identifiers, ending with 0 as a terminal symbol. For example, the proof line `10 2 3 -4 0 2 5 7 9 0` indicates that by resolving clauses 2, 5, 7, and 9 we obtain clause 10, consisting of the literals 2, 3, and -4. Clauses from the original problem instance are represented in the same way, but possess an empty derivation.

In proofs generated this way, chains that share large parts of their derivations frequently occur. In a proof of the pigeon hole principle (PHP), e.g., the following two proof chains show up, which differ only by three clauses ($6 \leftrightarrow 17, 8 \leftrightarrow 18, 10 \leftrightarrow 19$):

```
20 -15 -31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 0
21 -15 -32 0 1 2 3 4 5 17 7 18 9 19 11 12 13 14 15 16 0
```

So, there are (in principle) good chances to produce a shorter proof for these two assertions. The general idea to produce shorter proofs is to merge common subproofs occurring in different proof chains, where a subproof consists of a common subset of clauses. By merging them, repetitions of identical subproofs may be avoided. The problem is to identify subproofs that may be factored out. We want to illustrate the problem by an example:

Example 2. Assume two proof chains as shown below.

$$\dfrac{\dfrac{\{x, a\} \quad \{\neg a, b\}}{\{x, b\}} \quad \{\neg b, c\}}{\{x, c\}} \qquad\qquad \dfrac{\dfrac{\{y, a\} \quad \{\neg a, b\}}{\{y, b\}} \quad \{\neg b, c\}}{\{y, c\}}$$

Here, it is possible to factor out the common subproof

$$\dfrac{\{\neg a, b\} \quad \{\neg b, c\}}{\{\neg a, c\}}$$

[1] *booleforce* is available for download at http://fmv.jku.at/booleforce.

by re-ordering the given resolution proofs so that the clauses $\{\neg a, b\}$ and $\{\neg b, c\}$ are resolved first. Then the resolvent $\{\neg a, c\}$ is used to prove $\{x, c\}$ using $\{x, a\}$ and $\{y, c\}$ using $\{y, a\}$. Whereas the original proof chains require four resolution steps, the proof with the subproof factored out needs only three.

As a second example, consider the two proof chains

$$\frac{\dfrac{\{e, \neg a, \neg c\} \quad \{\neg a, c\}}{\{e, \neg a\}} \qquad \{a, b\}}{\{e, b\}} \qquad \text{and} \qquad \frac{\dfrac{\{f, \neg a, \neg c\} \quad \{\neg a, c\}}{\{f, \neg a\}} \qquad \{a, b\}}{\{f, b\}}\ .$$

Here, it is not possible to factor out a common subproof. A possible candidate for merging would be the subproof

$$\frac{\{\neg a, c\} \quad \{a, b\}}{\{b, c\}}\ ,$$

but when this proof is factored out, the desired clauses may not be re-derived (without introducing additional resolutions), as, e.g., resolving $\{b, c\}$ with $\{e, \neg a, \neg c\}$ produces the weaker clause $\{e, \neg a, b\}$, but not the desired result $\{e, b\}$.

4.1 The Merging Criterion

So we are left with the question, when it is possible to factor out a common subproof of two proof chains. We propose to base this decision not on the proof chains, but on the conflict graphs, as the conflict graphs allow for different resolution orders depending on the topological ordering of the nodes. We therefore define the notation of an *isolatable subgraph* of a conflict graph, which corresponds to a resolution subproof that may safely be factored out. The definition is based on a partial ordering $\succ_G$ on the nodes of a conflict graph, which is defined as follows: $n_1 \succ_G n_2$ iff there is a path from node n_1 to node n_2 in the conflict graph G.

Definition 1. *A subgraph S of a conflict graph G is called an* isolatable subgraph *of G, when all of the following holds:*

1. *If n_1, n_2 are nodes of S with $n_1 \succ_G n_2$, then $n' \in S$ for all $n_1 \succ_G n' \succ_G n_2$.*
2. *There is a unique minimal node m (w.r.t. $\succ_G$) in S.*
3. *m is the only node with edges leaving S (i.e. ending in $G \setminus S$).*

Criteria 1 and 2 ensure that all clauses of S can be resolved. Criterion 3 ascertains that no intermediate resolvent in S is used in other parts of the proof outside of S. We can now formulate our merging criterion:

Lemma 1. *Let G_1, G_2 be conflict graphs and S a common subgraph of G_1 and G_2. The resolution proof corresponding to S can be factored out, if S is an isolatable subgraph of both G_1 and G_2.*

Here, by common subgraphs we understand subgraphs that coincide in structure as well as node labeling (literals and clauses). Lemma 1 is the basis of our merging algorithm, which works as follows:

Input: A refutation proof Π of a formula F produced by a DPLL-style algorithm with clause learning.
1. Select two proof chains γ_1, γ_2 from Π.
2. Compute conflict graphs G_1 and G_2 for γ_1 and γ_2.
3. Determine all common maximal isolatable subgraphs $\{S_1, \ldots, S_k\}$ of G_1 and G_2.
4. Factor out the resolution proofs for $\{S_1, \ldots, S_k\}$ and adapt γ_1 and γ_2 accordingly.
5. Continue with 1., until no further common subgraphs can be extracted.
Output: Compressed proof Π'.

We use a heuristic in Step 1 of this algorithm to find suitable candidates for merging. This heuristics just considers the "neighborhood" of a clause (the n follow-up clauses in the proof) and tries to maximize the number of common clauses in the chains' derivations. It works well, as similar proof chains typically occur proximate to each other.

5 Implementation and Experimental Results

We have implemented the proof compression algorithm of the previous section in a C++ command line tool called *ProofCompress*. *ProofCompress* runs under both Linux and Mac OS X and is available from the author of this paper upon request. Besides compressing proofs, it can also dump conflict graphs in a format suitable for graph layout tools like AT&T's GraphViz (http://www.graphviz.org). The graph dump feature also allows to highlight common isolatable subgraphs of two proof chains.

Table 1. Results obtained with *ProofCompress* on various SAT instances.

Instance	original #resolutions	#resolutions after 100 compr. rounds	#resolutions after 1000 compr. rounds	runtime 100 rounds	runtime 1000 rounds
aim-200-2_0n4	119	110 (92.4%)	110 (92.4%)	0.03	0.03
dubois20	773	622 (80.5%)	608 (78.7%)	0.24	1.00
dubois30	1,291	1,053 (81.6%)	1,021 (79.1%)	0.39	1.98
dubois50	1,703	1,364 (80.1%)	1,316 (77.3%)	0.61	3.78
een-pico-p01-75	207,274	177,136 (85.5%)	160,852 (77.6%)	1576.39	2517.36
hole6	6,257	5,776 (92.3%)	5,122 (81.9%)	1.21	11.23
hole7	38,475	37,795 (98.2%)	36,094 (93.8%)	7.91	67.38
hole8	223,165	222,198 (99.6%)	220,123 (98.6%)	87.49	449.11
ibm-04-26k25	935	894 (95.6%)	894 (95.6%)	0.84	4.76
ibm-04-6023k100	256,858	196,433 (76.5%)	171,578 (66.8%)	255.80	756.17
longmult3	7,122	4,674 (65.6%)	4,266 (59.9%)	1.84	14.18
longmult4	32,811	22,392 (68.3%)	16,380 (49.9%)	8.86	45.76
longmult5	150,443	130,909 (87.0%)	92,446 (61.5%)	80.47	325.32
manol-pipe-g6bi	736,758	708,803 (96.2%)	584,510 (79.3%)	1567.02	3330.61
mutcb10	134,319	131,492 (97.9%)	121,449 (90.4%)	33.78	227.11
ssa0432-003	679	511 (75.3%)	511 (75.3%)	0.32	0.42

Table 1 displays results obtained with *ProofCompress* on an Intel Core 2 Duo processor running at 2.4 GHz under Linux. The first column shows the instance name (instances are available from http://fmv.jku.at/sat-race-2006 and http://www.satlib.org), the second column the number of resolution steps in a proof generated by *booleforce*. We used this proof as input to *ProofCompress* and let it run for 100 resp. 1000 compression rounds on each instance. Proof size (in number of resolutions) and run time for

ProofCompress under these two settings are shown in the remaining four columns. We observed proof compression ratios of up to 49.9% in our experiments.

6 Related and Future Work, Conclusion

Other work on obtaining smaller propositional proofs, besides what is mentioned in the introduction, was done by H. Amjad [20]. His proof compression method works on complete refutation proofs, and compresses them by reordering resolution steps. As it works on complete proofs, it may not scale to larger instances as well as our method.

We have presented an algorithm to compress proofs as generated by DPLL-style SAT solvers with clause learning. Our method works by merging similar proof chains. Besides having the potential to compress existing proofs, we suppose that our method may also be built into existing SAT-solvers to directly produce smaller proofs.

References

1. M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7, 1960.
2. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7), 1962.
3. J. P. Marques-Silva and K. A. Sakallah. GRASP — a new search algorithm for satisfiability. In *Proc. ICCAD'96*.
4. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC'01*.
5. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proc. DATE'02*.
6. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT'03*.
7. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS'99*.
8. M. Velev and R. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *J. Symb. Comput.*, 35(2), 2003.
9. I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Proc. ASE'03*.
10. C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1), 2003.
11. K. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Proc. TACAS'03*.
12. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proc. POPL'05*.
13. K. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV'03*, volume 2725 of *LNCS*.
14. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proc. DATE'03*.
15. A. Biere. Booleforce, 2006. Available at http://fmv.jku.at/booleforce.
16. T. Jussila, C. Sinz, and A. Biere. Extended resolution proofs for symbolic SAT solving with quantification. In *Proc. SAT'06*.
17. P. Beame, H Kautz, and A Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22, 2004.
18. N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Proc. SAT'06*.
19. J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12, 1965.
20. H. Amjad. Compressing propositional refuations. In *Proc. AVoCS'06*, 2006.