Clausal Proof Compression *

Marijn J.H. Heule[1] and Armin Biere[2]

[1] Department of Computer Science, The University of Texas at Austin, USA marijn@cs.utexas.edu
[2] Institute for Formal Models and Verification, JKU Linz, Austria biere@jku.at

Abstract

Although clausal propositional proofs are significantly smaller compared to resolution proofs, their size on disk is still too large for several applications. In this paper we present several methods to compress clausal proofs. These methods are based on a two phase approach. The first phase consists of a light-weight compression algorithm that can easily be added to satisfiability solvers that support the emission of clausal proofs. In the second phase, we propose to use a powerful off-the-shelf general-purpose compression tool, such as bzip2 and 7z. Sorting literals before compression facilitates a delta encoding, which combined with variable-byte encoding improves the quality of the compression. We show that clausal proofs can be compressed by one order of magnitude by applying both phases.

1 Introduction

Propositional proofs of unsatisfiability come in two flavors: resolution proofs [12] and clausal proofs [10]. An important drawback of using such proofs is their size on disk. Since resolution proofs can be up to two orders of magnitude larger compared to clausal proofs [5], the issue is much more severe for resolution proofs. Even clausal proofs are still too big for some applications, such as computing Van der Waerden number $W(2,6)$ [8] and the optimal sorting network with ten wires [2]. This paper offers some compression techniques to make them more compact.

The compression techniques presented in this paper are inspired by the binary variant of the AIGER format [1], the input format of the hardware model checking competition. This binary format stores the gates of sequential circuits using a binary representation instead of ASCII characters. Additionally, delta encoding is applied to store the difference between successive numbers. Sorting literals in a clause does not influences validity of proof, but reduces these differences between successive literals — making it a useful pre-compression technique.

Proof compression has many applications. For instance, a restriction to 100GB disk space, the maximal local storage on cluster nodes in the SAT 2014 competition[1], prevented the validation of some proofs of the unsatisfiability tracks. This can be avoided by adding a light-weight compression algorithm to SAT solvers to reduce the size proof lines written to disk. Notice that the 100 GB space limit was per benchmark per solver. Storing all unsatisfiability proofs of the competition is unfeasible even after strong compression.

*This work was supported by the Austrian Science Fund (FWF) through the national research network RiSE (S11408-N23) and the National Science Foundation under grant number CCF-1526760.

[1] results of the certified unsatisfiability tracks at http://satcompetition.org/2014

Clausal proof compression techniques are also useful to store proofs of hard combinatorial problems, such as the Erdős Discrepancy Conjecture (EDP) [7], for which a clausal proof is available. The initial proof was 13GB in size. Using symmetry-breaking methods, a proof of 2GB was produced [4]. In this paper, we show this proof can further be compressed to 128MB (less than 1% of the original proof). For other hard combinatorial problems, such as Van der Waerden numbers and minimal sorting networks, the expected size of (uncompressed) clausal proofs is many terabytes. Compression techniques will be crucial to deal with such proofs.

2 Preliminaries

CNF Satisfiability. For a Boolean variable x, there are two *literals*, the positive literal x and the negative literal $\bar{x}$. A *clause* is a disjunction of literals and a CNF formula a conjunction of clauses. A truth assignment is a function τ that maps literals to $\{\mathbf{f}, \mathbf{t}\}$ under the assumption $\tau(x) = v$ if and only if $\tau(\bar{x}) = \neg v$. A clause C is satisfied by τ if $\tau(l) = \mathbf{t}$ for some literal $l \in C$. An assignment τ satisfies CNF formula F if it satisfies every clause in F.

Resolution and Extended Resolution. The resolution rule states that, given two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \ldots \vee b_m)$, the clause $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, can be inferred by resolving on variable x. We say C is the *resolvent* of C_1 and C_2. For a given CNF formula F, the *extension rule* [9] allows one to iteratively add definitions of the form $x := a \wedge b$ by adding the *extended resolution clauses* $(x \vee \bar{a} \vee \bar{b}) \wedge (\bar{x} \vee a) \wedge (\bar{x} \vee b)$ to F, where x is a new variable and a and b are literals in the current formula.

Unit Propagation. The process of *unit propagation* simplifies a CNF F based on unit clauses. It repeats the following until fixpoint: if there is a unit clause $(l) \in F$, remove all clauses containing literal l from set $F \backslash \{(l)\}$ and remove literal $\bar{l}$ from all clauses in F. If unit propagation on formula F produces complementary units (l) and $(\bar{l})$, we say that unit propagation *derives a conflict* and write $F \vdash_1 \epsilon$ with ϵ referring to the (unsatisfiable) empty clause.

Example Consider $F = (a) \wedge (\bar{a} \vee b) \wedge (\bar{b} \vee c) \wedge (\bar{b} \vee \bar{c})$. We have $(a) \in F$, so unit propagation removes $\bar{a}$, resulting in the new unit clause (b). After removal of $\bar{b}$, two complementary unit clauses (c) and $(\bar{c})$ are created. From these two units the empty clause can be derived: $F \vdash_1 \epsilon$.

Clause Redundancy. A clause C is called *redundant* with respect to a formula F iff $F \wedge \{C\}$ is satisfiability equivalent to F. *Asymmetric tautologies*, also known as *reverse unit propagation* clauses, are the most common redundant (learned) clauses in SAT solvers. Let $\overline{C}$ denote the conjunction of unit clauses that falsify all literals in C. A clause C is an asymmetric tautology with respect to a CNF formula F iff $F \wedge \overline{C} \vdash_1 \epsilon$. *Resolution asymmetric tautologies* (or RAT clauses) [6] are a generalization of both asymmetric tautologies and extended resolution clauses. A clause C has RAT on $l \in C$ (referred to as the *pivot* literal) with respect to a formula F if for all $D \in F$ with $\bar{l} \in D$, it holds that $F \wedge \overline{C} \wedge (\overline{D} \setminus \{(l)\}) \vdash_1 \epsilon$. Not only can RAT be computed in polynomial time, but all preprocessing, inprocessing, and solving techniques in state-of-the-art SAT solvers can be expressed in terms of addition and removal of RAT clauses [6].

Clausal Proofs. A *proof of unsatisfiability* (also called a *refutation*) is a sequence of redundant clauses containing the empty clause. It is important that the redundancy property of clauses can be checked in polynomial time. A *DRAT proof*, short for *Deletion Resolution Asymmetric Tautology*, is a sequence of addition and deletion steps of RAT clauses. A DRAT *refutation* is a DRAT proof that contains the empty clause. Figure 1 shows an example DRAT refutation.

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

DRAT proof

```
  -1  0
d -1 -2  3 0
d -1 -3 -4 0
d -1  2  4 0
   2  0
   0
```

Figure 1: Left, a formula in DIMACS CNF format, the conventional input for SAT solvers which starts with `p cnf` to denote the format, followed by the number of variables and the number of clauses. Right, a DRAT proof for that formula. Each line in the proof is either an addition step (no prefix) or a deletion step identified by the prefix "`d`". Spacing in both examples is used to improve readability. Each clause in the proof should be an asymmetric tautology or a RAT clause using the first literal as the pivot.

Example Let $F = (a \lor b \lor \bar{c}) \land (\bar{a} \lor \bar{b} \lor c) \land (b \lor c \lor \bar{d}) \land (\bar{b} \lor \bar{c} \lor d) \land (a \lor c \lor d) \land (\bar{a} \lor \bar{c} \lor \bar{d}) \land (\bar{a} \lor b \lor d) \land (a \lor \bar{b} \lor \bar{d})$, shown in DIMACS format in Fig. 1 (left), where 1 represents a, 2 is b, 3 is c, 4 is d, and negative numbers represent negation. The first clause in the proof, $(\bar{a})$, is a RAT clause with respect to F because all possible resolvents are asymmetric tautologies:

$$\begin{array}{lll} F \land (a) \land (\bar{b}) \land (c) \vdash_1 \epsilon & \text{using} & (a \lor b \lor \bar{c}) \\ F \land (a) \land (\bar{c}) \land (\bar{d}) \vdash_1 \epsilon & \text{using} & (a \lor c \lor d) \\ F \land (a) \land (b) \land (d) \vdash_1 \epsilon & \text{using} & (a \lor \bar{b} \lor \bar{d}) \end{array}$$

3 Proof Compression

We propose to compress clausal proofs using two phases. The first phase is a light-weight method which can easily be added to any SAT solver that can produce proofs of unsatisfiability. The second phase applies a strong off-the-shelf compression tool to the result of the first phase.

Byte Encoding The ASCII encoding of clausal proofs in Figure 1 is easy to read, but rather verbose. For example, consider the literal `-123456789`, which requires 11 bytes to express (one for each ASCII character and one for the separating space). This literal can also be represented by a signed integer (4 bytes). If all literals in a proof can be expressed using a signed integer, only 4 bytes are required to encode each literal. Such an encoding also facilitates omitting a byte to express the separation of literals. Consequently, one can easily compress a clausal ASCII proof with a factor of roughly 2.5 by using a binary encoding of literals.

In case the length of literals in the ASCII representation differs a lot, it may not be efficient to allocate a fixed number of bytes to express each literal. Alternatively, the *variable-byte encoding* [11] can be applied, which uses the most significant bit of each byte to denote whether a byte is the last byte required to express a given literal. The variable-byte encoding can express the literal `1234` (`10011010010` in binary notation) using only two bytes: **1**1010010 **0**0001001. (in little-endian ordering, e.g., least-signifiant byte first).

Sorting Literals The order of literals in a clausal proof does not influence validness of the proof, nor does the order influences its size. However, the order of literals can influence the costs to validate a proof as it influences unit propagation and in turn determines which clauses will be marked in backward checking (the default validation algorithm used in clausal proof checkers). The order of literals in the proof produced by the SAT solver is typically not better or worse than any permutation. Experience shows that this is often not the case for SAT solving: the given order of literals in an encoding is generally superior compared to any permutation.

Sorting literals before compression has advantages in both phases. In the first phase, one can use delta encoding: store the difference between two successive literals. Clauses in a proof are typically long (dozens of literals) [5], resulting in a small difference between two successive sorted literals. Delta encoding is particularly useful in combination with variable-byte encoding.

In the second phase, off-the-shelf compression tools could exploit the sorted order of literals. Many clauses in proofs have multiple literals in common. SAT solvers tend to emit literals in a random order. This makes it hard for compression tools to detect overlapping literals between clauses. Sorting literals potentially increases the observability of overlap which in turn could increase the quality of the compression algorithm.

Table 1: Eight encodings of an example DRAT proof line. The first two encodings are shown as ASCII text using decimal numbers, while the last six are shown as hexadecimals using the MiniSAT encoding of literals. The prefix s denotes sorted, while the prefix ds denotes delta encoding after sorted. 4byte denotes that 4 bytes are used to represent each literal, while vbyte denotes that variable-byte encoding is used.

encoding	**example** (prefix pivot $\text{lit}_1 ... \text{lit}_{k-1}$ end)	**#bytes**
ascii	`d 6278 -3425 -42311 9173 22754 0\n`	33
sascii	`d 6278 -3425 9173 22754 -42311 0\n`	33
4byte	`64 0c 31 00 00 c3 1a 00 00 8f 4a 01 00 aa 47 00 00 c4 b1 00 00 00 00 00 00`	25
s4byte	`64 0c 31 00 00 c3 1a 00 00 aa 47 00 00 c4 b1 00 00 8f 4a 01 00 00 00 00 00`	25
ds4byte	`64 0c 31 00 00 c3 1a 00 00 e8 2c 00 00 1a 6a 00 00 cb 98 00 00 00 00 00 00`	25
vbyte	`64 8c 62 c3 35 8f 95 05 aa 8f 01 c4 e3 02 00`	15
svbyte	`64 8c 62 c3 35 aa 8f 01 c4 e3 02 8f 95 05 00`	15
dsvbyte	`64 8c 62 c3 35 e8 59 9a d4 01 cb b1 02 00`	14

Literal Encoding In most SAT solvers, literals are mapped to natural numbers. The default mapping function $map(l)$, introduced in MiniSAT [3] and also used in the AIGER format [1] converts signed DIMACS literals into unsigned integer numbers as follows:

$$map(l) = \begin{cases} 2l + 1 & \text{if } l > 0 \\ -2l & \text{otherwise} \end{cases}$$

Table 1 shows a DRAT proof line in the conventional DIMACS and in several binary encodings. For all non-ASCII encodings, we will use $map(l)$ to represent literals. Notice that the first literal in the example is not sorted, because the proof checker needs to know the pivot literal (which is the first literal in each clause). The remaining literals are sorted based on their $map(l)$ value.

4 Experiments

We implemented two tools: `ratz` (encode) and `ztar` (decode)[2]. We used `ratz` to transform DRAT proofs in the ASCII format to several alternative representations and applied off-the shelf compression tools to make the resulting files more compact. The compression tools used during the experiments are `gzip`, `bzip2`, and `7zip`. Due to space limitations the experiments focus on a single proof: a trimmed (i.e., removed redundant lines) DRAT proof [3] for Erdős Discrepancy

[2] available at http://fmv.jku.at/ratz

[3] available at http://www.cs.utexas.edu/~marijn/sbp

Problem [7] based on symmetry-breaking [4]. We selected a trimmed proof, because in practice one wants to remove redundancy before compression.

Table 2 shows the results. The second column shows that delta encoding combined with sorting and variable-byte encoding (last row of the table) reduces proof size by already more than a factor of four in 25 seconds. This significant and efficient compression can easily be added to any SAT solver that can produce clausal proofs, thereby reducing the space burden. The results of the second phase, i.e., using off-the-shelf compression tools, are less clear. The smallest file is produced by sorting and variable-byte encoding followed by 7zip. Delta encoding, although it reduces the size in combination with variable-byte encoding, appears to be obstruct all the compression tools.

Table 2: Size of a trimmed DRAT proof (in bytes) and the conversion costs (wall-clock time in seconds) for Erdős Discrepancy Problem using different encodings and compression algorithms. A four core Intel Xeon E31280 @ 3.50GHz with 32GB memory was used for the experiments. The tool 7z used all cores, while the other programs used only a single core.

encoding	**first phase**	**gzip**	**bzip2**	**7z**
ascii	1,719,002,352 (——)	224,505,003 (58.68)	186,871,192 (183.63)	176,740,892 (173.44)
sascii	1,719,002,352 (48.75)	199,368,062 (51.43)	153,589,408 (204.46)	155,268,644 (162.71)
4byte	1,282,405,483 (19.46)	205,093,278 (47.75)	182,221,318 (98.61)	163,176,124 (114.07)
s4byte	1,282,405,483 (27.28)	179,853,433 (39.46)	144,742,387 (116.32)	141,086,084 (109.31)
ds4byte	1,282,405,483 (27.07)	210,994,395 (49.49)	168,958,717 (86.05)	157,274,204 (121.58)
vbyte	639,781,147 (12.76)	183,079,542 (24.70)	183,254,546 (58.39)	149,944,476 (66.89)
svbyte	639,781,147 (20.07)	158,535,823 (22.72)	146,177,432 (63.44)	128,300,756 (66.69)
dsvbyte	403,398,345 (17.97)	157,295,747 (16.15)	165,947,521 (45.96)	136,576,424 (40.42)

5 Conclusion

We proposed several compression techniques for clausal proofs. In particular the combination of delta and variable-byte encoding is very useful to make proofs more compact. Both techniques can easily be added to SAT solvers, which would hardly increase the costs to emit a clausal proof. Off-the-shelf compression tools can be used to further reduce the proof size. Combining both phases on a proof of Erdős Discrepancy Problem shows a compression of over 93%.

References

[1] Armin Biere. The AIGER and-inverter graph (AIG) format, version 20070427, 2007.

[2] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In *ICTAI 2014*, pages 186–193. IEEE Computer Society, 2014.

[3] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[4] Marijn J. H. Heule, Jr. Hunt, Warren A., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *CADE-25*, volume 9195 of *LNCS*, pages 591–606. Springer, 2015.

[5] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design*, pages 181–188. IEEE, 2013.

[6] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 355–370. Springer, 2012.

[7] Boris Konev and Alexei Lisitsa. A SAT attack on the Erdős Discrepancy Conjecture. In *SAT 2014*, volume 8561 of *LNCS*, pages 219–226. Springer, 2014.

[8] Michal Kouril and Jerome L. Paul. The van der Waerden number W(2, 6) is 1132. *Experimental Mathematics*, 17(1):53–61, 2008.

[9] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2*, pages 466–483. Springer, 1983.

[10] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008.

[11] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2Nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1999.

[12] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885, 2003.