Harder SAT Instances from Factoring with Karatsuba and Espresso

Joseph Bebel
Department of Computer Science
University of Southern California
Los Angeles, California, United States of America
joseph.bebel@gmail.com

***Abstract*—These benchmarks were generated by the latest version of the ToughSAT software and are based on the hardness of factoring products of prime numbers. Instances based on products of primes are always satisfiable by exactly 2 assignments and, assuming widely accepted cryptographic assumptions, should be computationally hard to solve, with superpolynomial difficulty in the size of the primes used. The latest version of ToughSAT includes substantial improvements to the absolute and asymptotic complexity of the produced SAT instances, producing instances with significantly fewer variables and clauses for the same size prime numbers.**

***Index Terms*—factoring, SAT, karatsuba multiplication, fast integer multiplication, circuit optimization, boolean formula optimization**

I. Introduction

The ToughSAT project (https://toughsat.appspot.com/) by Joseph Bebel and Henry Yuen is designed to be an easy-to-use and easy-to-understand open source (https://github.com/joeintheory/ToughSAT) generator of computationally hard SAT instances. Specifically, it is a concrete implementation of several conceptually well-known strategies for hard SAT instance generation packaged as a web app and python script allowing quick and easy generation.

Integer Factoring is a well known problem in NP widely assumed to be hard in the average-case. This problem is not often necessary to compute in most practical applications and is most widely associated with cryptographic applications where instance sizes large enough to be computationally impractical to solve.

While ToughSAT is not the first software to generate hard SAT instances from Integer Factoring, a major goal of the project is to produce the "hardest possible" SAT instances from each Integer Factoring instance. Ideally, large instances of Integer Factoring (as measured by the number of bits in each prime factor) should be practically convertible to reasonably large SAT instances. For example, cryptographic size instances should produce instances which are not excessively large as to be unwieldy and impractical to generate, store, and load into SAT solvers (whether they are solvable by a SAT solver is a separate issue). In principle, assuming the the Exponential Time Hypothesis, solving SAT instances should require worst case running time exponential in the number of variables. Therefore, it is reasonable to suspect that "not many" additional variables are necessary to encode Integer Factoring into SAT instances.

The author gratefully acknowledges the support of the Annenberg Graduate fellowship and the ARCS Foundation Los Angeles Chapter.

ToughSAT has been improved towards this goal where smaller instances now produce more efficient boolean formulas using fewer variables and clauses in smaller DIMACS files. Interestingly, the reduction in variables anecdotally seems to produce instances which take longer to solve (compared to less efficient SAT instances generated from the same prime numbers).

II. Improvements to ToughSAT

The primary improvements to ToughSAT occurred in 2015 (the original ToughSAT was released in 2011) and have come from the use of the Karatsuba multiplication algorithm and the Espresso heuristic logic minimizer (https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/espresso/index.htm), along with some additional small optimizations.

The Karatsuba multiplication algorithm is a "fast integer multiplication" algorithm capable of multiplying n bit integers in time $O(n^{\log_2 3})$ time which is asymptotically better than the $O(n^2)$ time of classical integer multiplication. For our purposes, even though the numbers we are multiplying are rather small (20-30 bits per multiplicand, to obtain SAT instances that are solvable in practice) using the Karatsuba algorithm does produce smaller instances than the classical multiplication algorithm. While there are fast integer multiplication algorithms asymptotically superior to the Karatsuba algorithm, their large constant overhead likely would produce SAT instances larger than Karatsuba's in the 20-30 bit integer regime. It remains an open question whether asymptotically better integer multiplication algorithms would be superior in the cryptographic regime (greater than 384 bits per multiplicand), although such SAT instances are likely completely impractical as SAT solver benchmarks.

The instances are further shrunk substantially by use of the Espresso logic minimizer. The minimizer is not used directly in the construction of each SAT instance but rather was used to find minimal or at least smaller "building blocks" of each boolean functional unit used in Karatsuba's algorithm. For example, the initial implementation of Karatsuba's algorithm

may use basic units such as single bit adders, used in the form of CNF formulas on 5 variables that are satisfied on valid input/outputs of a 1 bit binary adder. This results in many intermediate variables, which for our purposes are wasteful. Instead, we can encode entire functional units (such as 8 bit adders and 6 bit multipliers) in CNF directly, with only the external input/output variables required (the internal intermediate variables being removed). Such CNF formulas of an 8 bit adder or 6 bit binary multiplier are rather large if written out in standard product of sums format (essentially describing the truth table of those functions) but can be optimized quite substantially with Espresso. The Espresso optimized functional units are used in our implementation of Karatsuba's algorithm.

In the small prime regime we are most interested in (at most 30 bit primes) this means we can avoid a substantial number of intermediate variables. For comparison, the 30 bit primes instance generated here needed only 988 intermediate variables to produce the 60 bit product, not far off from the approximately 900 intermediate variables required to simply even write down the partial products of the classical integer multiplication algorithm (completely ignoring the complex addition steps afterward).

III. Instances Generated

We generated a test instance constructed from two 23 bit prime numbers, using ToughSAT version 20190413, and used CryptoMiniSat 5.6.8 on default settings to estimate difficulty. CryptoMiniSat found a satisfying assignment in 313.2 seconds on a MacBook Pro (13-inch Retina Late 2013) with Intel Core i7-4558U 2800 MHz dual core CPU. Therefore, problems in this regime (22 to 26 bits) should be feasibly solvable using state-of-the-art SAT solvers given substantial CPU time.

28 distinct prime numbers (14 pairs of equal sized primes) were generated: 2 pairs of 22 bit primes; 4 pairs of primes of each of 23, 24, 25, and 26 bits; 1 pair of each of 27, 28, 29, and 30 bits. The primes were generated by OpenSSL 1.1.1b with the command "openssl prime -generate -bits b" where b is the number of bits required. Each pair was used to generate a single instance "20190413_toughsat_$b$bits_i.dimacs", where b is the number of bits in each prime factor and i is the index of the pair used. Each pair of primes uses distinct primes and no instances share prime factors.

In each DIMACS file, the target number (the product of the two primes) is given, along with the indexes of the variables encoding each prime factor. The input bits are encoded so that least-significant bit has smaller variable, so for example $p = x_{30}x_{29}\dots x_1$ and $q = x_{60}x_{59}\dots x_{31}$.

An observation: the 30 bit instance is nowhere near cryptographic size, however it may still be impractically large for some SAT solvers. The instance itself is quite reasonably sized (1108 variables, 27721 clauses, 897061 byte DIMACS file). In theory, subexponential time algorithms for Integer Factorization can solve products of 30 bit primes using almost no computational resources. However, to our knowledge there is no SAT solver in existence which implicitly or explicitly "encodes" the knowledge of number fields or other number theory to exploit any structure of the Integer Factoring problem. Therefore, it is not immediately obvious how exhaustive search can be avoided except for a constant number of bits. It is an interesting question to evaluate the extent to which a SAT solver can reliably find exploitable structure in factoring-sourced instances.

IV. Conclusion

The goal of the ToughSAT project is to use conceptually well known and understood methods to produce small, hard, SAT instances, as both an educational tool and a competitive hard SAT instance generator. Although by now, the optimizations and implementation details of ToughSAT are quite complex, the basic structure is still conceptually simple to describe using a widely understandable toolkit of concepts (integer factoring, fast integer multiplication) to produce competitively small and hard SAT instances. The reduction of intermediate variables has interesting effects on the perceived hardness of each instance. It would be interesting if a SAT solver could somehow reconstruct the functional units used in Karatsuba's algorithm to simplify its own analysis of each instance, or even implement higher order number theoretic methods.