Using co-SAT solvers to find lower bounds on detecting triangles

Josh Burdick

July 15, 2016

Abstract

Detecting whether an undirected graph contains a triangle (a.k.a. 3-clique, or K_3) is a tiny case of the NP-complete CLIQUE problem. We would therefore expect it to be both hard to answer quickly, and also difficult to *prove* hard to answer quickly. Here, we encode the problem of finding a NAND gate circuit which detects triangles, as a CNF formula (in DIMACS-CNF format, suitable as input for SAT or co-SAT solvers). This provides a possible source of either lower bounds, or difficult co-SAT instances (although it's not clear which).

The problem of detecting triangles has been studied, partly because it is a tiny special case of detecting cliques, which is NP-complete. In circuit terms, triangles can be detected using "brute force", using $\binom{n}{3}$ AND gates (one per possible triangle), and an OR gate to combine the outputs of those; or alternatively with $\binom{n}{3}+1$ NAND gates.

The triangle detection problem can also essentially be solved by multiplying two Boolean matrices together [1]. Successively faster algorithms allow doing this in $O(n^{2.8074})$ [2] or $O(n^{2.3727})$ [3] operations. The big-O notation hides large constants, and the "operations" counted in analysis of those methods are multiplications, which can involve large numbers. However, to multiply n-by-n Boolean matrices, we can use addition and multiplication mod $n+1$ (FIXME cite CLR), avoiding computing large numbers. Thus, such matrix multiplication algorithms allow triangle detection which asymptotically use fewer than $\Omega(n^3)$ NAND gates. [1] Nonetheless, finding bounds on the number of NAND gates needed to detect triangles in small graphs may be interesting.

Previously, I wrote up an attempt at a proof that finding triangles in 6-input graphs requires $\binom{6}{3}+1=21$ NAND gates [4]. (The proof has not been thoroughly checked, or published.) I tried pushing the same methods to get a proof for the $n=7$ case, but they didn't seem to be going much higher than "slightly more than the number of inputs." (I don't think this is very surprising.)

[1] In a previous version of this, I didn't realize this, and so thought that the $\binom{n}{3}+1$ circuit might be optimal; I now realize that it isn't.

Therefore, I decided to try applying brute force, using solvers for propositional logic. Development of such solvers has been accelerated by their use in logic circuit design. I first tried encoding the problem as a quantified Boolean formula, and, using a BDD solver, looking for a circuit which finds triangles (or, more importantly, ruling out the existence of such a circuit). Since that was using a large amount of memory, I then tried using a SAT (or co-SAT) solver to look for a circuit (or rule out the possibility of a circuit) which only *approximately* finds triangles.

0.1 Motivation

Why might bounds on circuits for triangle-finding in tiny graphs be useful? Since we're only looking at tiny input graphs (and circuits), such bounds have limited relevance to the $P = NP$ question.

It seems natural to guess that "you need at least one NAND gate per triangle", and try to prove that. (This conjecture seems related to the Strong Exponential-Time Hypothesis, or SETH [5].) (However, as noted earlier, using fast multiplication methods, you don't need at least one NAND gate per triangle.)

The encoded formulas may also be useful as test problems for SAT solvers. Showing that a formula is not satisfiable is an important problem, and benchmarks and competitions for solvers of such problems often require a source of test problems known to be unsatisfiable, and/or difficult to solve [6]. Assuming $P \neq NP$, a formula corresponding to the existence of a small circuit for an NP-complete problem should be unsatisfiable. (To reduce the problem size, we relax the requirement that the circuit must work on all possible inputs; this means that the formula we write may be satisfiable, even if no circuit with that many gates *exactly* finds triangles.)

Note that this is somewhat win-win: either the generated formulas are easy to prove unsatisfiable (in which case we get better lower bounds on triangle finding), or they're hard to prove unsatisfiable (in which case we have more probably-unsatisfiable problems for testing co-SAT provers; admittedly this may be less of a win.)

1 Encoding as a quantified Boolean formula

One convenience of looking for bounds on NAND gate circuits (as opposed to, say, circuits with AND, OR, and NOT gates, or arbitrary Boolean functions) is that the gates are all the same kind. The circuit is defined only by its connectivity graph.

This means that we can encode a circuit using a Boolean variable for each wire which might be present. Specifically, we can encode an m-input circuit made of g NAND gates using mg bits (one per input-gate wire), plus $\binom{g}{2}$ bits (one per gate-gate wire). We also have one variable per input wire (edge), and can code up a formula for "has a triangle." To code up the formula "there is a circuit which works", we have to universally quantify over the inputs (because the circuit has to work on all inputs). This causes a blowup in the formula size.

When I ran this, with four vertices and five gates, it fairly quickly found a working circuit. [2] For five vertices, it ruled out a circuit with five gates. I tried larger numbers of gates (six or seven, I don't recall which, but definitely less than the known lower bound of eleven gates), and it just allocated memory until it crashed my machine (a Thinkpad X200s with 8gb memory.)

1.1 Forcing wires to be sorted

I also tried adding a constraint that the rows of the connectivity matrix are in lexicographic order. We can do this without loss of generality. Proof sketch: pick an arbitrary order for the input wires. For all gates which only have front-level inputs, write their inputs as a bit-vector, find the lexicographically first, and call it g_1. Then find g_2 by only considering front-level inputs, and (possible) wires from g_1, treating it as the most significant bit. Repeat until you've added all the gates. This defines a unique ordering of gates, such that the adjacency matrix of all wires is lexicographically sorted.

This may have helped a bit, but not much (data not shown; I look at the effect of this more carefully below.)

2 Approximate triangle-finding circuits

In the previous attempt, we quantified over all the possible inputs. This meant that the notion of whether there was a triangle was part of the formula. As in Razborov's monotone circuit bound [7], we can also approximately specify how the circuit should work. MAX-CLIQUE, at least, is hard to approximate, unless $P = NP$ [8], so we may not have to specify every possible input graph to force the circuit to have many gates. Since the resulting formula is just Boolean (without quantifiers), I used a SAT solver [9] rather than a BDD solver.

This approach is reminiscent of machine learning: we are asking for a "simple" circuit which explains some examples. At first, I used the $\binom{n}{3}$ cases which have one triangle (and zeros elsewhere) as positive examples. Following Razborov's monotone circuit bound [7], I used complete bipartite graphs as negative examples.

This small number of examples was enough to show that, with four vertices, at least five gates were needed. However, with five vertices, there were circuits which distinguished those test cases, but had fewer than $\binom{5}{2} + 1 = 11$ gates. Therefore, I added "test cases" which were random graphs, and added a constraint that the circuit would output 0 or 1 (according to whether the graph contained a triangle.)

[2] I didn't actually print out the solution for $n = 4$, with five gates, so it could be incorrect. Given the small numbers involved, I'm not planning on bothering with this further.

3 Results

For a small set of tiny problems, the following table shows whether the problem was satisfiable (and there is a circuit with that many gates that is correct, at least given the number of sample inputs given). Also shown is the runtime, on a lightly-loaded X200s laptop. (Where "satisfiable?" is -, I stopped the program by hand.)

num. vertices	num. gates	num. random cases	wires sorted?	satisfiable?	CPU time
4	2	0	no	no	0 s
4	4	0	yes	no	0 s
4	5	0	no	yes	0 s
4	5	0	yes	yes	0.004 s
5	11	30	no	yes	57.696 s
5	11	30	yes	yes	53.656 s
5	5	0	no	no	0.02 s
5	5	0	yes	no	0.012 s
5	7	0	no	yes	0.16 s
5	7	0	yes	yes	0.152 s
5	7	30	no	no	0.752 s
5	7	30	yes	no	0.884 s
5	8	30	no	no	23.936 s
5	8	30	yes	no	38.788 s
5	9	10	no	yes	1.624 s
5	9	10	yes	yes	0.628 s
5	9	20	no	yes	19.248 s
5	9	20	yes	yes	38.816 s
5	9	30	no	-	606.724 s
5	9	30	yes	-	608.624 s

4 Conclusions

This approach didn't improve the lower bounds. However, it was less memory-intensive than using BDDs. It may be a useful source of "likely to be unsatisfiable" SAT instances. Forcing the wires' adjacency matrix to be sorted didn't help much.

4.1 Future work

Rather than using random graphs as training data, it might be possible to choose those more systematically. Possibly graphs with multiple triangles would be better positive examples. Or, perhaps two-edge paths (which are triangles with an edge removed) would be better negative examples.

Boiling the problem down to propositional logic presumably loses a lot of the inherent symmetry in the problem. However, fully automated SAT solvers (that is, zero-order theorem provers) are arguably more developed than fully automated first- or higher-order theorem provers.

References

[1] Alon Itai and Michael Rodeh. Finding a Minimum Circuit in a Graph. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 1–10, New York, NY, USA, 1977. ACM.

[2] Prof Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

[3] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *In Proc. 44th ACM Symposium on Theory of Computation*, pages 887–898, 2012.

[4] Finding triangles in 6-vertex graphs requires 21 nand gates. Available online at `http://www.cs.umd.edu/~gasarch/BLOGPAPERS/burdick1.pdf` (part 1) and `http://www.cs.umd.edu/~gasarch/BLOGPAPERS/burdick2.pdf` (part 2).

[5] Russell Impagliazzo and Ramamohan Paturi. The Complexity of k-SAT. In *2012 IEEE 27th Conference on Computational Complexity*, volume 0, page 237, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[6] ToughSAT Generation. Available online at `https://toughsat.appspot.com/`.

[7] A. A. Razborov. Lower bounds on the monotone complexity of some boolean functions. Dokl. Akad, 1985.

[8] Johan Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182(1):105–142, March 1999.

[9] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, number 2919 in Lecture Notes in Computer Science, pages 502–518. Springer Berlin Heidelberg, May 2003. DOI: 10.1007/978-3-540-24605-3_37.