Bounds on finding triangles (in *tiny* graphs) with NAND gates

Josh Burdick

July 31, 2008

Finding cliques is one problem which has been studied for lower bounds. It's hard to approximate, and Razborov showed that a monotone (AND- and OR- gate) circuit which solves it has exponentially many gates. Therefore, if you're going to look for lower bounds, it seems like a worthwhile problem to look at.

How many (unbounded fan-in) NAND gates are needed to find triangles in an undirected graph? Here, we focus on *tiny* graphs, of up to six vertices. This is, of course, a teensy special case. However, it is a well-defined number, and at least the question "does this require at least $\binom{6}{3} + 1$ gates?" isn't one of those wifty questions which, due to Gödel's theorem, might be neither true nor false.

I'm explicitly not thinking about cases when $n > 6$ or $k > 3$, because presumably that's difficult...

0.1 Preliminaries and definitions

The input is an n-vertex undirected graph. The actual input lines to the circuit will correspond to the $\binom{n}{2}$ possible edges of the graph. For each pair of vertices, we input a 1 to the circuit if that edge is present, and 0 otherwise. We expect an output of 1 from a properly-functioning circuit iff there's a k-clique in the input graph.

Throughout this discussion, we only deal with the case where $k = 3$, that is, the case of finding triangles in the input graph.

The *output gate* is the last gate, whose output is supposed to be 1 iff there's a clique. Most of the time, we don't count the output gate, because none of the counting methods here will directly count it.

An *input gate* is any gate directly connected to one or more inputs.

Let C be a circuit made of unbounded-fan-in NAND gates. A *subcircuit* of C is the circuit, formed by just using some subset of the inputs, and feeding in 0's elsewhere. (Note that a gate can be in lots of subcircuits.) Any gate whose output becomes constant (either directly because of receiving a 0, or more indirectly) is not considered part of the subcircuit.

A *triangle subcircuit* is a subcircuit formed by feeding in 0s to everything except one triangle's edges.

We'll number the input graph's vertices from 1 through 6.

0.2 An upper bound

The "obvious" circuit to find triangles using AND and OR gates uses one AND gate per triangle, plus an OR gate to combine the results. This adds up to $\binom{n}{3}$ AND gates and one OR gate.

If we change all the gates in the above circuit to NAND gates, it still works correctly (although all of the inputs to the output gate are inverted.) Thus, $\binom{n}{3} + 1$ NAND gates suffices to find triangles in an n-vertex graph.

It is difficult to imagine how to improve on this circuit; it seems that at least one gate is needed for each triangle. However, this doesn't prove anything.

1 A lower bound of $\binom{n}{2} + 1$

First, note that no input wire can be connected directly to the output gate. (Proof: suppose it were. Then if you input all 0s to the circuit, it outputs 1, although there aren't any cliques there.)

Consider the $\binom{n}{3}$ triangle subcircuits (that is, the subcircuits formed by feeding in 0s to everything except one triangle.) Note that this disables many of the gates in the circuit, because as soon as you input a 0 to a NAND gate, it will always output a 1, regardless of the rest of its inputs, and so isn't doing anything useful. (We only count the non-constant gates in each subcircuit.)

In each of these subcircuits, at least one wire must be directly connected to some non-output gate. (Clearly, each subcircuit must look at all of its inputs. And they can't be connected to the output gate.)

We count, in each subcircuit, the number of wires from edge inputs to input gates. There must be at least three (one for each edge of the triangle.)

We're possibly overcounting here, though, because a given gate could occur in more than one subcircuit.

Note that any gate with direct connections to more than one triangle isn't in any triangle subcircuit. (If a gate is directly connected to gates in two subcircuits A and B, then when we are looking at subcircuit A, that gate is getting a 0 from some edge in subcircuit B. This means that when we look at subcircuit A, that gate's output is fixed at 1, it's not doing anything useful, and indeed isn't even in subcircuit A (because our definition excludes gates which are constant.)

An input gate can be connected to one, two, or three of the edges of a triangle (and possibly gates from other circuits as well), and will be in that triangle's subcircuit. However, if it's connected to two or three edges, it can only be in that triangle's subcircuit. (Because if it's connected to, say, edges 12 and 13, then it's in triangle subcircuit 123, but in subcircuit 124, it's getting a 0 on edge 13. Similarly when it's connected to three edges.)

Let a_k be the number of non-output gates in C with k direct connections to inputs. If we draw all the subcircuits separately, then a gate with only one direct input could be in up to $n-2$ subcircuits. (It can only be in subcircuits whose triangles contain that input wire.) But a gate with more than one direct input can only be in one subcircuit (the subcircuit for the triangle which contains all its input wires.)

Counting those direct connections, we get that

$$(n-2)a_1 + 2a_2 + 3a_3 \geq 3\binom{n}{3} \tag{1}$$

This implies that $\sum a_k$, the total number of gates, is ≥ 4 when $n = 4$, and ≥ 10 when $n = 5$. These bounds match the upper bounds (though it doesn't show that the $n = 5$ circuit only has 3-direct-input gates.)

In general, according to this bound, a circuit can get away with making a_1 large – that is, making all input gates have only one direct input. Then

$$\begin{aligned}
(n-2)a_1 &\geq 3\binom{n}{3} \\
(n-2)a_1 &\geq 3\frac{n(n-1)(n-2)}{6} \\
a_1 &\geq \frac{n(n-1)}{2}
\end{aligned}$$

$$a_1 \geq \binom{n}{2}$$

Of course, I'd imagine that a circuit with all input gates having only one input connection would require many more gates. Unfortunately, this bound doesn't help with that. In general, you could add a layer of $2\binom{n}{2}$ double-NOT gates to any circuit, and this bound won't "see" anything past the first layer, so it doesn't seem pushable past $\Omega(\binom{n}{2})$.

2 Improving that bound when $n = 6$

In the above bound, it intuitively looks like a_1 is what we really have to worry about. The gates directly connected to two or three inputs cover two or three input wires, but can only be in one subcircuit.

The gates connected to only one input still can be in lots of subcircuits; when $n = 6$, such gates can be in four subcircuits at once, and thus cover four input wires. (As an example, if all the front-level gates are one-input gates, i.e. NOT gates, there are only fifteen non-output gates, but certainly all the inputs are covered.) So the above method really isn't going to go higher than the number of input wires.

Call a gate *standard* if it's just connected to all three inputs of a triangle.

In this section, I'll assume that all the input gates are connected to at most one input. This is an unfortunate gap, which means that the results in this section aren't proven for all NAND gate circuits. However, it seems to be the hardest case, since these are the gates which can "be in more than one subcircuit at once", and so are harder to count.

2.1 Using a straightline circuit

As is pretty standard (certainly Razborov's monotone circuit bound used this), we assume the circuit can be arranged in a line, with all wires going forward – a *straightline circuit*. (Which had better be possible, since the circuit's a DAG.)

For now, we assume that no gate is directly connected to more than one input. Unfortunately, I don't know how to avoid this assumption.

We will now feed in 0s and 1s to the edges, starting with whatever edge the first gate is connected to.

Note that the first gate must be a one-input NAND gate – a NOT gate, in other words. If we feed it a 1, it will output a 0. This 0 had better not go to the output gate (if it did, the output would be stuck at 1 even if the rest of the inputs were 0, and there were no triangle.) Therefore, at least one other gate must read this 0, and effectively be disabled.

Also note that whenever we feed in a constant to the circuit, we can remove gates which are now constant, and this process can be repeated. If at any point of this "constant propagation" the first gate in the circuit doesn't have any directly-connected input wires, then its output must be a constant, and we can propagate its output. When we're done, the first gate in the circuit must *behave* like a NOT gate, even if it's not (because it has other inputs.)

If it's behaving like a NOT gate, we can feed in a 1. Once again, this forces it to output a 0, which again must be used somewhere, which disables another gate. Thus, we've counted another two gates.

So, we repeatedly feed in a 1 or 0, and simplify the circuit, see which edge is now first, and continue with that edge. At each point, we need to make sure that the gate we're feeding in a 1 or 0 to "matters": there must be some assignment to the remaining inputs such that when a 1 is fed in, the circuit's output is 1, and when a 0 is fed in, the circuit's output is 0.

We can keep feeding in 1s, and thus "knocking out" an extra gate, but only if we haven't formed a triangle, or made it impossible for there to be a triangle. (In either of those cases, the circuit would have been reduced to a constant.)

2.2 Order of feeding in 1's

When feeding in 1's, we don't get to choose which edge will be considered next; we need the gate with that input to be a NOT gate, or acting like one, and which gate is determined by the circuit's straightline ordering. However, we can feed in 0's whenever, and as long as this edge is part of some triangle, we know we'll knock out some gate.

Not knowing the order of edges is a real pain. If I could feed in 1's in any order, I'd feed in 1's in a complete bipartite graph on two sets of three vertices, which would be $9 \cdot 2 = 18$ gates. Then I'd feed in 0's to the remaining edges, all of which would still be relevant, since if I had fed in a 1, there would be a triangle. (This is like one part of Razborov's bound.)

Unfortunately it doesn't seem to be so simple.

2.3 One strategy: write out the game tree

You can look at this as a game in which the circuit tells you which edge is next, and you feed in a 1 or a 0 to that edge.

If at any point, the input graph contains a triangle of all 1's, you lose (because the circuit now will definitely output a 1, and all the other edges might be irrelevant.) If the input graph ever contains an edge such that all triangles incident to it contain at least one 0, then you lose, because you can no longer say anything about gates connected to that edge.

If you feed in five 1's (and some number of 0's), and haven't violated those conditions, then you win.

The question is, can you win, no matter what the order of edges is?

The game isn't quite that hard, however, because although the order of feeding in 1's is dictated by the circuit, we can feed in a 0 whenever, so long as the edge we're feeding it to is still definitely connected to a gate. So, after we feed in a 1, we might as well count the edges which *would* form a triangle if that edge was next in the circuit, by feeding in 0's to them.

If you do this, I think the game tree indicates that you win. (I can send this if you're curious. But I think the strategy below is a better bet, because it seems to deal better with the case in which there's a mixture of 1-direct-input and 2- or 3-direct-input gates.)

2.4 Another strategy

Here is another attempt at a strategy for feeding in 0s and 1s. Set aside three triangles, say 123, 345, and 561. Call the edges in those triangles "triangle edges", and the other edges "middle edges".

The strategy has two phases.

In the first phase, feed in 0's to the middle edges, one at a time. We know that each of those edges hasn't been zonked because for each middle edge, there's a triangle consisting of that edge, and two triangle edges. (Proof: every such edge connects two of the triangles 123, 345, or 561. Each pairof those triangles contains a shared vertex, either 1, 3, or 5. The middle edge, plus the triangle edges to the shared vertex, together form a triangle.) This will count six gates.

In the second phase, we treat the remaining gates as a straightline circuit, and feed in 1's, until a given triangle would have all 1's, in which case we feed in a 0. This gives $2+2+3=7$ 1's, plus two 0's, for a total of $14+2=16$

gates.

Adding these, we would get a bound of $6 + 16 = 22$ NAND gates.

2.4.1 The bug, and a possible workaround

The problem with that is: when you're feeding in 1's in the second phase, the order of edges might be 13, 35, 15, in which case you'd get the triangle 135. To avoid this, you could feed in 0 to edge 15. But then how do you count the edges 15 and 16 (since all the triangles they would have been part of have at least one zero fed into them)?

If you could just zero out the edges in one of those three triangles (say, 561), then you'd avoid this triangle 135 (and count three gates.) But then, after you feed in the first 0, the other two edges aren't in any triangle, and so aren't provably connected to any "live" edge.

Then again, by assumption, the input gates are only connected to at most one input. In the triangle 561, we can show that there's a one-input gate connected to edge 56, by feeding in 1's to edges 15 and 16, and noting that the input to edge 56 matters. A similar argument holds for 15 and 16. So we know there are three gates there, and can feed in three 0's, and remove them from consideration. I think this fixes it.

3 Gates connected to more than one input?

So far, in *all* of the previous section, we assumed all the input gates only connected to one input. What if there are, say, standard 3-direct-input gates? If all the subcircuits are standard, then clearly there are $\binom{n}{3}$ of them.

But what if there's a mixture of standard, and nonstandard gates? Note that to cover one input edge more efficiently than 1-direct-input gates, you need four 3-direct-input gates (for subcircuits 123, 124, 125, and 126, to cover edge 12, for instance.)

We can feed in 0's to "knock out" some of these. For instance, if some edge has four standard gates connected to it, then we can feed in a 0 to that edge, and knock out four gates at once.

It looks like if, after feeding in 0's, there are at least two triangles with disjoint edges left over, with *only* 1-direct-input gates, then we can use the above argument to count an additional $2 \cdot 5 + 1$ gates. However, I haven't worked out the details of this.

4 Other facts

4.1 Standard circuits at a vertex

Call a subcircuit *standard* iff it consists only of one NAND gate, connected to all of its inputs. In other words, it uses one gate to find each triangle.

Consider a circuit C, which finds triangles in 5-vertex graphs. Further, suppose that all of the subcircuits of C containing some vertex v_i are standard. Then we can delete that vertex, feeding in zeros to all lines connected to it. This will remove at least $\binom{5}{2} = 10$ gates (these are "easy to count" because all the subcircuits containing v_i are "standard.") The circuit which remains finds triangls in 5-vertex graphs, so by the previous bound has at least 10 non-output NAND gates. Therefore, under those assumptions, C must have at least twenty non-output NAND gates.

Thus, in order to do better than the all-standard circuit, *each* of the six vertices must have *some* edge which leads to a non-standard subcircuit. Each non-standard subcircuit can only account for at most three distinct vertices, so there must be at least two non-standard subcircuits in C, which are connected to completely disjoint sets of vertices.

It means that a circuit which improves on the standard circuit must be "nonstandard everywhere".

4.2 Number of standard circuits

Another similar fact is that any working circuit, that has fewer than twenty non-output gates, can't have more than ten standard circuits.

A given edge can be covered either by a 1-direct-input gate, or by four 2- or 3-direct-input gates (one for each triangle subcircuit), or by a mixture. If an edge is covered, say, by a 3-direct-input gate and a 1-direct-input gate, then the 1-input gate is only "in three places at once", instead of being "in four places at once".

Let g_s be the number of standard subcircuits. Let g_1 be the number of nonstandard subcircuits. $g_s + g_1 = 20$.

Let e_1 be the number of edges in all the triangles with one-input inputs. I think, without proving it, that $e_1 \geq g_1$, at least when the number of vertices is six, and there are up to ten triangles. (In other words, g_1 distinct triangles, which may share edges, have at least g_1 distinct edges. For example, if there

are 10 triangles, then the fewest edges there could be is 10, if the triangles form a 5-clique.)

At any rate, if you believe that, then the total number of gates at least the number of standard gates, plus the total number of edges covered by 1-direct-input gates, which is

$$g_s + e_1 \geq g_s + g_1 = g_s + (20 - g_s) = 20$$

This means that, to beat the standard circuit, there can't be very many standard subcircuits (there must be fewer than ten), which is reminiscent of the previous result.

4.3 Smaller circuits

Consider a circuit C. Pick a vertex in the input graph, and feed in 0's to all input lines to C from edges which include that vertex. The original circuit found triangles, so those input lines connected to some non-output gates in C, so the resulting circuit has strictly fewer gates.

This is unsurprising, and doesn't seem that helpful.

4.4 Distinct circuits

It's clear that non-overlapping subcircuits must have distinct gates. (Proof: feed in all 0's to one of them. This removes some gates, which aren't present in the other subcircuit, because that subcircuit is still finding triangles.)

If two distinct subcircuits overlap, then in the case of triangles, this can only happen if they share an edge. Call the triangles 123 and 124 (so they share edge 12.) Feed in a 1 to edge 12. When you feed in 0's to 13 and 23 (thus removing some gates), the subcircuit for 124 is still finding triangles (edge 12 is fixed at 1, but it's still using 14 and 24.) Therefore, the subcircuit for triangle 124 contains some non-output gates which aren't in the subcircuit for triangle 123.

Again, not very surprising, but not so helpful. Showing that each subcircuit contains at least one particular gate that's not in any of the other subcircuits remains quite tricky. (For instance, given n elements, it's fairly simple to construct $2^{n/2}$ sets, such for each pair of sets, one of them contains something the other doesn't.)

4.5 Nesting of subcircuits

Let C be a subcircuit given by feeding in 0's to all but some set A of vertices. If you feed in 0's from some vertex, then you'll definitely cause some gates to stop doing anything.

So, we can define a function f from sets of vertices to sets of gates. If $A \subset V$ is a set of vertices, $f(A)$ is the set of gates in the subcircuit made by feeding in 0's to all edges incident to an edge not in A. If $|A| < 3$, then $f(A)$ is empty, but if $|A| \geq 3$, then $f(A)$ is definitely non-empty.

For any subset $B \subset A$, $f(B) \subset f(A)$. (This reminds me of the definition of a "continuous function" from topology, not that I really know much topology.) Indeed, if $B \subset A$ and $B \neq A$, then $f(B) \subset f(A)$ and $f(B) \neq f(A)$. So the subcircuits are strictly nested.

This seems unlikely to be helpful, because even if you were using AND, OR, and NOT gates, probably you could still prove somehow that feeding in 0's to all edges incident to a vertex eliminated at least one gate; and so someone probably would have done this. It might be harder, though.

5 Conclusions

The good news is that this bound is actually slightly larger than the number of inputs. Party! But seriously, consider doing this bound with AND, OR, and NOT gates. If I had tried that, it would be slightly trickier to show that no input line is connected to the last gate. Also, "feeding in 0's" would no longer definitely render a gate useless, because it could be an OR gate, or a NOT gate. In general, there would be three cases to worry about at each step, instead of one. So I would argue that this is an improvement over dealing with AND, OR, and NOT gates. (It might reduce a hypothetical proof that clique-finding requires exponentially many NAND gates from 300 pages, to a mere 294.)

The bad news is, this bound is basically counting the number of inputs, and using NAND gates doesn't seem to help much. The feeding-in-1's method is kind of nice, but you certainly can't feed in more 1's then there are inputs to the circuit, so it won't get you that far.