Finding triangles in 6-vertex graphs requires 21 NAND gates (DRAFT)

Josh Burdick

January 29, 2009

This is an attempt at proving what the title says. It's not written up rigorously. However, I think the goal is modest enough that there's some hope that it's correct. Even if it isn't, it may either be close, or provide some useful or interesting ideas.

I assume that you've read my previous attempt at proving this.

0.1 Notation

A *subcircuit* is the circuit obtained by feeding in 0's to all input edges, except for one triangle. Note that a given gate can be in more than one subcircuit at once.

A subcircuit is *covered by* a gate/gates if, when you zonk a circuit down to just that subcircuit, those are the front-level gates which remain. Clearly, each subcircuit will have (at least) three input wires connected to it.

A gate is *zonked* if it's been fed a 0. When I say, for instance, "zonk a circuit down to just a subcircuit", I mean "feed in 0's to everything except the three input wires in the subcircuit."

A *k-direct-input gate* is a gate with exactly k input wires connected to actual edge inputs (and possibly with input wires connected to other gates as well.)

The last gate or *output gate* is never affected by any of the feeding in 0's or 1's, so I'll just talk about "number of gates required" instead of saying "number of non-output gates required."

A *standard gate* is a two- or three-direct-input gate. One of these can only be in one subcircuit, since outside of one subcircuit, they must be getting a 0 on some input wire, and are consequently zonked.

A *standard subcircuit* is a subcircuit which contains at least one standard gate. A subcircuit can be covered by all sorts of numbers of gates, of course. However, as soon as a subcircuit contains at least one standard gate (whether it has two or three inputs), you know that that gate is only in that subcircuit.

1 Bounds when there are *lots* of one-direct-input gates

As noted earlier, the one-direct-input gates can be in many subcircuits at once. We must charge the circuit for this, at least a little bit...

1.1 A subcircuit covered only by one-direct-input gates requires five NAND gates

As seen in the previous attempt, if at some point you end up with a subcircuit with only three inputs, all of which are only covered by 1-input gates, then you know that that subcircuit requires at least five NAND gates. The proof is by the aforementioned induction, feeding in 1's when possible. Specifically, by feeding in 1's to the first two edges which show up at the front of the circuit (knocking out two gates each time), and a 0 to the final edge (zonking one more gate.)

It could, for instance, contain exactly five non-output gates. It could just be a three-input gate connected to one edge directly, and through double-NOTs on the other two inputs. (Since the three-input gate isn't directly connected on all three wires, it's not a standard subcircuit.)

1.2 *Two* edge-distinct subcircuits covered by one-input gates require 10 NAND gates

If, at some point, the only non-constant inputs are two non-overlapping triangles, and each is "covered by a non-standard circuit" (that is, only has 1-input wire front-level gates), then the same basic idea still works. You feed in two 1's to each triangle (counting two gates each time), and then feed in a 0 to the last edge (counting one more gate.)

The 1's may show up all in one triangle first, or alternate, or whatever, but the induction still works. Before the inputs to a triangle have all been set, it's possible that any of its remaining inputs could complete a triangle, and so all of those inputs are important. After the inputs have all been set (to two 1's and lastly a 0), that subcircuit won't be part of any triangle, and so even if one triangle has had its two 1's and a 0 fed in, it won't affect the proof that the other subcircuit requires five NAND gates.

The two triangles can share no vertices, or one vertex, but not an edge. Also, if feeding in 1's to one of them creates a triangle in the other, then the circuit's output is now fixed at 1, and you can't count any more gates. This might come up, say, if the triangles were 123 and 145, and the edge 24 had been fed a 1 already. So, if you feed in 1's earlier, you have to make sure they can't form a triangle with these others.

Note that in the previous paper, my attempt to have three subcircuits each require five NAND gates was broken. At the time, I said that was fixable, but I no longer believe that. This is because the edges could come up in an order that creates a triangle between the three triangles. As far as I can tell, there's no easy fix for this; in a sense, a lot of this discussion is working around this.

1.3 Getting rid of the standard gates

The 1-direct-input gates, of course, can show up in several subcircuits (up to four.) The above proof, that each non-overlapping triangle requires at least five gates, makes this more expensive, at least. However, that proof requires that, as you feed things into the straightline circuit, you always end up with a 1-direct-input gate (not counting all the constants it's received so far) at the front of the circuit. If you don't, then you can't guarantee that the current earliest gate will output a 0 (and thus knock something else out.)

So, the simplest thing seems to be to get rid of *any* standard gates before we get to filling in the two triangles.

1.4 Removing standard gates

We can knock out all the standard gates first, by feeding in 0's. Suppose, for instance, that the circuit contains just two non-standard subcircuits 123 and 456, and the other subcircuits are covered by standard three-input gates. (So, there are six one-direct-input gates, connected to edges 12, 13, 23, 45, 46, 56, and nine other three-direct-input standard gates.) If we feed in nine 0's to everything except the edges which are connected to at least one 1-direct-input gate, then eighteen standard gates are safely zonked. Even though they all have some input wires from 123 and 456, they also have at least one from outside those triangles, which will zonk them when a 0 is fed in there.

We don't have to worry about having an exact one-to-one correspondence between 0's fed in, and gates zonked. However, the gates besides those in subcircuits 123 and 456, in this case, certainly aren't guaranteed to be standard gates, alas.

2 Bounds for different numbers of standard subcircuits

The above example gave a high enough bound when there were *lots* of standard subcircuits. We need a high enough bound, no matter how many standard subcircuits (< 20) there are.

To do this, we break things down by the number of standard subcircuits. I will list these in order, from most to fewest standard subcircuits, because the cases get progressively harder.

If there are 20 standard subcircuits, of course, we're done, because a standard subcircuit contains at least one (non-output) gate, and a standard gate can only be in one subcircuit. Hopefully the other cases will all require at least 21 gates...

2.1 10 to 19 standard subcircuits

If there are between 10 and 19 standard subcircuits, then there are between 10 and 1 non-standard subcircuits. These non-standard subcircuits only contain 1-direct-input gates.

First, we need this lemma.

2.1.1 For $1 \leq n \leq 10$, n triangles have at least n edges

To prove this, I ended up brute-forcing this: looping through all $2^2 0$ subsets of triangles (in six-vertex graphs), and checking that at least that many edges were included. The C code is included in the last section. The program computes, for each number of triangles, the minimum number of edges which can be covered.

triangles	minimum number of edges
0	0
1	3
2	5
3	6
4	6
5	8
6	9
7	9
8	10
9	10
10	10
11	12
12	13
13	13
14	14
15	14
16	14
17	15
18	15
19	15

This basically makes sense. When there's one triangle, at least three edges are covered. For up to 10 triangles, there are at least as many edges as triangles; they can't be "packed" any more closely than in a five-vertex clique. There's probably a cleaner way to prove this (maybe by counting the average number of triangles touching an edge.) I'm not worrying about this for now.

2.1.2 The total number of gates is ≥ 20

There are 20 subcircuits. Let c_3 be the number of standard subcircuits, and c_1 be the number of non-standard subcircuits. Every subcircuit is either standard or non-standard, so $c_3 + c_1 = 20$.

Each of the standard subcircuits contains one gate which isn't in any of the other subcircuits, standard or otherwise.

$1 \leq c_1 \leq 10$, so by the above lemma, the non-standard subcircuits include at least c_1 edges. Each of these needs a 1-direct-input gate to cover it, which means that there are at least c_1 1-direct-input gates. Each standard subcircuit contains at least one (2- or 3-direct input) gate, which isn't in any other subcircuit. Adding up the gates from the standard subcircuits, plus those in the non-standard subcircuits, is $c_3 + c_1 = 20$ gates.

2.1.3 The total number of gates is ≥ 22

So far, we haven't ruled out some circuit with exactly 20 gates, but which looks totally unlike the "standard" circuit with 20 3-input gates.

To do this, find a non-standard subcircuit A. Use the previous argument to find at least 20 gates (a mix of standard and non-standard gates, in general), with at least one input wire, and put them in a set S. Now feed in 0's to all edges except those in the triangle inputting to A. There are at least three gates from S, still live in A.

Now, use the "triangle has at least five gates" lemma to count five gates in the subcircuit A, by feeding in two 1's and a 0. Three of these you'll already have counted, and are in S. However, the two gates which were zonked when you fed in a 1 can't have been in S (if they had been, they would have had an input wire connected to somewhere besides the edges in A.)

Adding the 20 gates in S, plus the two additional gates, is 22.

2.2 2 to 9 standard subcircuits

If there are between 2 and 9 standard subcircuits, then we can find two non-standard subcircuits which share no vertices. To see this, note that there are ten pairs of vertex-disjoint triangles (123 and 456, 124 and 356, etc.) There are no more than nine standard subcircuits, so even if each standard subcircuit hits one or the other of different pairs, there still will be one pair left.

Renumber the vertices so that 123 and 456 are non-standard subcircuits. We will feed in 0's to the other nine edges; I'll call them the *middle edges.*

Each of the middle edges is in four subcircuits, so 36 wires need to be covered. (Note that these wires can be counted more than once, in multiple subcircuits.) A 1-direct-input gate can cover four wires, since it can be in four subcircuits. But a 2- or 3-direct-input gate can only cover two wires, since that gate can only be in one subcircuit (and one of those edges is in 123 or 456, and doesn't help with these nine edges.)

2.2.1 2 or 3 standard subcircuits

If there are 2 or 3 standard subcircuits, then none of the middle edges is completely covered by standard subcircuits, and so each middle edge is also covered by a 1-standard-input gate. This makes for at least $2 + 9$ gates connected to the middle edges. We feed in 0's to the middle edges, leaving two edge-disjoint triangles; from those we get another $2 \cdot 5$ gates (see above.) All told, that's 21 gates.

2.2.2 4 to 9 standard subcircuits

Let g_1 be the number of 1-direct-input gates connected in the middle, and g_3 be the number of 2- or 3-direct-input gates. The nine middle edges are in four different subcircuits, so there are 36 wires which need to be counted. A 1-direct-input gate can cover four of these (one in each of four subcircuits), while a 2- or 3-direct-input gate can only cover at most two of them (in one subcircuit; the third wire is in 123 or 456, and so doesn't help.)

Since all 36 of these wires are covered,

$$4g_1 + 2g_3 \geq 36$$

$g_3 \geq 4$, so even if the remaining 28 wires are covered by 1-direct-input gates, $g_1 + g_3 \geq 4 + 7 = 11$. These 11, plus the ten from triangles 123 and 456, is 21 gates.

2.3 1 standard subcircuit

This is similar to the 0 standard subcircuits case (see below.) But you need to get rid of the standard subcircuit first, by feeding in a 0 to one of its edges, which zonks at least two gates (remember, the standard gate can only be in one subcircuit, so there must be at least one 1-direct-input gate also connected to it.)

As before, you can then definitely feed in two 1's, knocking out two gates with each. Having done this, you can still set aside two triangles, feed in 0's everywhere else, and still have ten gates left in the triangles. Picking the triangles is slightly hairier than when there are 0 standard subcircuits, though, as it depends on which 0 you choose, and where the first two 1's go.

For each possible arrangement of 1 0'ed-out edge and two more 1 edges, I need to find two edge-disjoint triangles such that even when 1's are fed into the triangles, it doesn't form additional triangles outside of those triangles. Fortunately there aren't two many distinct arrangements of three edges, and it doesn't matter which 1 edge is which.

0'ed edge	1'ed edges	triangles to set aside
12	13, 14	236, 456
12	13, 34	156, 245
12	13, 23	145, 256
12	34, 45	136, 256
12	34, 56	145, 236

All told, that's 1 for the first 0, $2 \cdot 2$ for the next two 1's, 6 for the next six 0's (not in the triangles), plus $2 \cdot 5$ for the two triangles. This is $1 + 4 + 6 + 10 = 21$ gates.

Personally, I'd call this the most hideous case.

2.4 0 standard subcircuits

In this case, all front-level gates are 1-direct-input. So, we'll run the previous feeding-in-1's algorithm twice, knocking out four gates. Since we've only fed in two 1's, we can't have formed a triangle yet, and since for every other edge, there's an input assignment which will "make that edge matter" (that is, force a 1 final output, by completing a triangle, if the edge is 1, and 0 otherwise), all the other edges still have working connections to (some) front-level gates.

Either the two 1's we fed in shared a vertex, or they didn't.

If they did, number them 12 and 23. Set aside subcircuits 345 and 561, and feed in seven 0's to the remaining seven gates, knocking out seven more 1-direct-input gates. What's left is two triangles; use the above lemma to find five more gates in each. (Note, crucially, that even with 12 and 23 set to 1, no adding 1's to 345 will cause a triangle in 561, unless of course all of 561 edges are 1's. All told, this means the circuit has $2 \cdot 2 + 7 + 2 \cdot 5 = 21$ gates.

Otherwise, if they don't share a vertex, number the edges 12 and 34. Set aside triangles 235 and 146. Feed in 0's to the other seven 1-direct-input gates. Then, as before, find five gates in each of the two triangles remaining. Again, the only possible triangles are 235 and 146, and the triangles are independent, and we find a total of 21 gates.

3 Conclusions

This seems to give a bound for the the $n = 6$, $k = 3$ case which matches the upper bound. Fan-in and depth are both unbounded.

Happily, all the cases (except the standard circuit) require strictly more than $\binom{6}{3}$ non-output gates. This prevents the possibility that the lower bound is 20 non-output gates, but there's some bizarre circuit, presumably with lots of one-direct-input gates, which also works, besides the standard circuit.

This proof seems unlikely to generalize. It's nice that the one-direct-input gates are the only ones that can be in multiple subcircuits. But they're only getting charged one additional gate.

This doesn't really do anything beyond the first level of gates. As the number of vertices n gets large, the front layer could be covered with $\binom{n}{2}$ one-direct-input gates, and the proof methods discussed here would not say much about the next level of gates in.

4 C code

This program considers all subsets of 20 triangles in a six-vertex graph, and counts how many of the 15 edges are covered by at least one triangle.

```
#include <assert.h>
#include <stdint.h>
#include <stdio.h>

/** Stores the bit masks for all 20 triangles. */
uint64_t tri[20];

/** The maximum number of edges, given the number of triangles. */
int minEdges[20];

/** Returns a uint64_t with the i'th bit set. */
uint64_t mask(int i) {
return 1 << i;
}

/** Counts the bits set in a uint64_t. */
```

```
int bitCount(uint64_t a) {
int sum = 0;
int i;
for(i=0; i<32; i++)
if (a & mask(i))
sum++;
return sum;
}

/** Returns the bitmask corresponding to some edge.
 XXX this wasn't working on 32-bit systems.  So I
just tweaked it to pack all the edge bits into no
more than 32 bits (as opposed to just using 8*i+j). */
uint64_t edgeBits(int i, int j) {
assert(i < j);
return mask(i + j*(j+1)/2);
}

/** Sets up the list of triangles. */
void initTri() {
int t = 0;
int i, j, k;

for(i=0; i<4; i++)
for(j=i+1; j<5; j++)
for(k=j+1; k<6; k++)
tri[ t++ ] =
edgeBits(i, j) |
edgeBits(i, k) |
edgeBits(j, k);
}

/** Computes the edges in a given set of triangles. */
uint64_t triEdges(uint64_t triSet) {
uint64_t edges = 0;
int i;

for(i=0; i<20; i++)
if (triSet & mask(i))
edges |= tri[i];

return edges;
}

/** Counts number of edges, and triangles. */
void count() {
uint64_t i;

for(i=0; i<mask(20); i++) {
int numTris = bitCount(i);
int numEdges = bitCount( triEdges( i ));
if (numEdges < minEdges[numTris])
minEdges[numTris] = numEdges;
}
}


int main(int argc, char **argv) {
int i;
for(i=0; i<20; i++)
minEdges[i] = 1000000;   /* or whatever */

initTri();
count();

printf("%10s %10s\n",
"triangles", "    minimum number of edges");
for(i=0; i<20; i++)
printf("%10d %10d\n", i, minEdges[i]);

return 0;
}
```