Lower bounds for the edge function: a readable account of ten Lean-checked statements

This document is a human-readable account of ten statements about the circuit size of the edge function f_G of a graph G (true exactly when both ends of some edge are true), each checked in Lean 4 with Mathlib. Where the prose and the Lean differ, the Lean is the authority. The prose itself is not machine-checked and has not been read by a human mathematician. The Lean files are public at https://github.com/brilliant-hq/pequalnp-proofs, under lean/PEqualNP/MetaComplexity/: one file per statement, named in the table of section 0, with the helper modules they import. The run column of that table names the checker run that accepted each file.

24 November 2026.

0. What is stated

All circuits have fan-in at most two over AND, OR and NOT; size counts every gate, NOT gates included; inputs are free. f_G(x) = OR over edges uv of (x_u AND x_v). tau(G) is the vertex cover number. A HUB is a vertex with at least three neighbours. A PAIR FUNNEL of a function is a pair of distinct inputs p, q at which three of the four restrictions to the values of (x_p, x_q) coincide (stated exactly in Theorem 3.5 and in 5.1, H2).

row	statement, in words	Lean	run
L-meta-204	G with every cycle of length >= 7 and no walk of length < 7 between two distinct vertices of degree >= 3; A a set of vertices that no edge meets; A or E nonempty: every circuit for (OR of x_a, a in A) OR f_G has at least |A| + |E| + tau(G) - 1 gates	`MetaComplexity/L204.lean`, `L204` := `EdgeFn.stmtLB_final`	run-013633
L-meta-104	G a forest with an edge: every circuit for f_G has at least |E| + tau(G) - 1 gates	`L104.lean`, `Helpers.Forest.forest_bound`	run-011343
L-meta-102	path on n inputs: a circuit with s gates exists iff n + floor(n/2) <= s + 2; so the size is n + floor(n/2) - 2 for n >= 2	`L102.lean`, `Helpers.PathSize.path_size`	run-011301
L-meta-106	cycle on n >= 5 inputs: a circuit with s gates exists iff n + ceil(n/2) <= s + 1	`L106.lean`, `Helpers.CycleSize.cyc_exact_of_readers` `L107`	run-012105
L-meta-105	any G with an edge, any vertex cover T: f_G has a circuit with |E| + |T| - 1 gates	`L105.lean`, `Helpers.EdgeUpper.edge_upper`	run-011310
L-meta-154	the same, with a circuit that has no NOT gate	`L154.lean`, `Helpers.EdgeUpperMono.edge_upper_mono`	run-013182
L-meta-115	cycles >= 9, hubs >= 7 apart: every circuit for f_G WITHOUT NOT gates has at least |E| + |T| - 1 gates for some vertex cover T	`L115.lean`, `Loc.Final.row`	run-012892
L-meta-113	cycles >= 9, hubs >= 9 apart: every circuit for f_G can be replaced by one with no NOT gate and no more gates	`L113.lean`, proof in the file	run-014387

row	statement, in words	Lean	run
L-meta-112 (section 6)	the 4-cycle's edge function: a circuit with s gates exists iff s >= 3	`L112.lean`, `Helpers.ExactCycle4.size_iff`	run-011875
L-meta-107 (used by 106)	a function that depends on some input, whose restrictions at every input it depends on are not constant, and that has no "pair funnel": from any circuit with s gates, fixing some input to false leaves a circuit with at most s - 3 gates, or fixing some input to true leaves one with at most s - 4 gates, the number of inputs kept	`L107.lean`, `Helpers.Readers.fanout_step_3_4`	run-011915

Consequence, read off the rows (not a row of its own; on paper, not checked, outside the class of L-meta-113, see the next sentence): with A empty, on forests with an edge and on the graphs of L-meta-204 with an edge, the least circuit size of f_G is exactly |E| + tau(G) - 1, NOT gates included, attained by a circuit with no NOT gate (lower bound L-meta-104 or L-meta-204 with k = tau(G); upper bound L-meta-154 with T a least cover). The only Lean proof that makes this combination is the proof of L-meta-113 in `L113.lean` (L-meta-204 at A empty with k the size of a least cover, and a copy of the L-meta-154 circuit), on the class of cycles at least 9 and hubs at least 9 apart; on forests and on the rest of the class of L-meta-204 the consequence is on paper, not checked.

1. Definitions

Below, `H/` is `lean/PEqualNP/MetaComplexity/Helpers/`.

1.1 Gates and circuits (root: `lean/PEqualNP/Defs/CircuitLowerBounds.lean`, `GateOp`, `Gate`, `Circuit`, `Circuit.wires`, `Circuit.eval`). A gate is an operation and a list of wire indices (repetitions allowed). The operations used here are AND (true on the empty list), OR (false on the empty list) and NOT (the negation of the head of the list; further entries ignored; true on the empty list). A circuit is a list of gates in evaluation order and an output index. On an input string x of length n the wires are x_0, ..., x_{n-1}, then one wire per gate: gate number j (counted from 0) produces wire n + j, the value of its operation on the wires it names. A wire index that does not exist at that moment reads false. The value of the circuit is its output wire, false if that wire does not exist. Acyclicity is built in: a gate sees only wires that already exist.

1.2 Circuit size (root: `Circuit.size`, same file). The number of gates. Inputs are not counted; NOT gates and constant gates (AND or OR of the empty list) are counted.

1.3 Fan-in two AND, OR, NOT circuits, and "has a circuit of size s" (root: `lean/PEqualNP/Defs/MetaComplexity.lean`, `FanInTwo`, `HasCircuitSize`; and `lean/PEqualNP/Defs/MetaComplexityMinimal.lean`, `circuitFun`). `FanInTwo C`: every gate of C is AND, OR or NOT and names at most two wires. For f : {0,1}^n -> {0,1}, `HasCircuitSize n f s`: some circuit C with `FanInTwo C`, size at most s, and `C.eval (List.ofFn x) = f x` for every x. `circuitFun n C` is the function x -> `C.eval (List.ofFn x)`. Since a missing output reads false, the constant false has size 0. The constant true has size 1: one gate, AND or NOT of no wire, computes it, and no circuit with 0 gates does, since its output is then an input wire or a missing wire, and either reads false at the all-false input. Where the paper notion is not a root definition: "the circuit size of f"

(the least such s) is not defined anywhere in the root. The rows state it through `HasCircuitSize`: an exact value v is written as "for every s, `HasCircuitSize n f s` iff v <= s" (L-meta-102, L-meta-106), an upper bound as `HasCircuitSize n f v` (L-meta-105) or as an explicit circuit D with `D.size + 1 <= v + 1` (L-meta-154). "No NOT gate" is spelled `forall g in D.gates, g.op != GateOp.not`.

1.4 Graphs and the edge function (not a root definition). A graph is a Mathlib `SimpleGraph (Fin n)` (symmetric, no loops) on the vertex set {0, ..., n-1}, which is also the set of input positions. The edge function is spelled in each row statement as `decide (exists i j : Fin n, G.Adj i j and x i = true and x j = true)`, that is

f_G(x) = OR over edges ij of (x_i AND x_j).

The number of edges is `G.edgeFinset.card` (L-meta-104, 105, 115, 154) or `G.edgeSet.ncard` (L-meta-204, and the helper potential below); both count the edges, written |E|. The path function P_n and the cycle function C_n are not stated through a graph: L-meta-102 spells P_n(x) = OR over i, j with i + 1 = j of (x_i AND x_j), and L-meta-106 spells C_n(x) = OR over i, j with (i + 1) mod n = j of (x_i AND x_j). P_n is the edge function of the path 0 - 1 - ... - (n-1) for every n; C_n is the edge function of the cycle on 0, ..., n-1 for n >= 3 (at n = 1 it is x_0, at n = 2 the single edge 01; the row L-meta-106 has n >= 5).

1.5 The edge function with extra OR inputs (not a root definition). For a set A of vertices,

F_{G,A}(x) = (OR over a in A of x_a) OR f_G(x).

L-meta-204 spells it in its statement as `@decide ((exists a in A, x a = true) or exists i j, G.Adj i j and x i = true and x j = true) _` with the hypothesis "no vertex of A is an end of an edge" (`forall i j, G.Adj i j -> i not in A`). The proofs use the helper `Forest.F G A` (`H/ForestBound.lean`), the same expression, and the helper `Loc.AFree G A` (`H/LocGraph.lean`) for the hypothesis. At A empty, F_{G,A} = f_G. Sections 2 and 3 below do not use A; it is defined here for the later sections.

1.6 Vertex covers and the vertex cover number (not root definitions as used). A vertex cover is a finite set T of vertices with `forall i j, G.Adj i j -> i in T or j in T`; every row writes this condition out. The root has `IsVertexCover` (`lean/PEqualNP/Defs/CoverPromise.lean`), the same condition, but none of these rows uses it. The vertex cover number tau(G) does not occur in any row statement. The lower-bound rows L-meta-204 and L-meta-104 take a number k with `k <= T.card` for every cover T (so the bound holds in particular at k = tau(G)); the lower-bound row L-meta-115 takes no k and instead produces some cover T with `G.edgeFinset.card + T.card <= D.size + 1`; an upper-bound row takes an explicit cover T (so the bound holds in particular at a cover of size tau(G)). The proofs use the helpers `Forest.Cov` (`H/ForestBound.lean`, the cover condition) and `Loc.tau` (`H/LocGraph.lean`, the infimum of the sizes of covers, attained by `Loc.tau_attained`).

1.7 The formula value. The quantity the rows bound is

|A| + |E| + tau(G) - 1 (and |E| + tau(G) - 1 at A empty).

No row subtracts on the lower-bound side: L-meta-204 states `A.card + G.edgeSet.ncard + k <= s + 1` and L-meta-104 states `G.edgeFinset.card + k <= s + 1`. L-meta-105 uses the natural-number expression `G.edgeFinset.card + T.card - 1` (no truncation occurs, since the graph has an edge); L-meta-154 states `D.size + 1 <= G.edgeFinset.card + T.card`. The helper `Loc.Phi1 G A` (`H/LocGraph.lean`) is |A| + |E| + `Loc.tau G`, the formula value plus one. On the path and the cycle the formula value is a closed expression: P_n has n - 1 edges and cover number floor(n/2), giving n + floor(n/2) - 2 (the rows write `n + n / 2 <= s + 2`); C_n has n edges and cover number ceil(n/2),

giving n + ceil(n/2) - 1 (the rows write `n + (n + 1) / 2 <= s + 1`, with (n + 1) / 2 = ceil(n/2) in integer division). The rows L-meta-102 and L-meta-106 do not mention covers at all; the identification with the formula value is ours, on paper.

2. The upper bound: the cover circuit

Theorem 2.1. Let n be a natural number, G a simple graph on {0, ..., n-1} with at least one edge, and T a set of vertices such that every edge has an end in T. Then the edge function f_G has a fan-in-two AND, OR, NOT circuit with at most |E| + |T| - 1 gates. Lean: `lean/PEqualNP/MetaComplexity/L105.lean`, `PEqualNP.MetaComplexity.L105`; checked as L-meta-105, run-011310.

Theorem 2.2. Under the same hypotheses there is a circuit D, every gate AND, OR or NOT reading at most two wires, with no NOT gate, computing f_G, with size(D) + 1 <= |E| + |T|. Lean: `lean/PEqualNP/MetaComplexity/L154.lean`, `PEqualNP.MetaComplexity.L154`; checked as L-meta-154, run-013182.

The two proofs are the same text in two files (`H/EdgeUpper.lean` for 2.1, `H/EdgeUpperMono.lean` for 2.2); 2.2 carries one more invariant, "no gate is NOT". We give them together.

Building at an offset (Lean: `Bld`). Say F : {0,1}^n -> {0,1} is BUILT with s gates if for every m >= n there is a gate list gs, every gate AND, OR or NOT of at most two wires (Lean: `Ok`), with at most s gates and an output index below m + |gs|, such that running gs after ANY wire list w of length m gives, at the output index, the value F(w_0, ..., w_{n-1}). (For 2.2, also: no gate of gs is NOT.) The arbitrary offset m is what lets two built functions be placed one after the other.

Step 1 (Lean: `bld_var`). An input x_i is built with 0 gates: the empty list, output index i.

Step 2 (Lean: `bld_bin`, with `bld_and`, `bld_or`). If F is built with s1 gates and H with s2, then F AND H and F OR H are built with s1 + s2 + 1. Proof: after m wires place a list for F; after the resulting m + |gs1| wires place a list for H (allowed, since H is built at every offset); append one gate AND (resp. OR) reading the two outputs. The gates for F are not disturbed by the later ones, and the list for H reads the inputs at their unchanged positions. The appended gate is AND or OR, so no NOT gate appears. Monotonicity in s and replacing F by an equal function are `bld_mono`, `bld_congr`.

Step 3 (Lean: `bld_anyList`, `bld_anyFinset`). The OR of the inputs of a nonempty set S is built with |S| - 1 gates: list S as v, w, ..., and build x_v OR (OR of the rest) by Step 2, by induction on the list.

Step 4 (Lean: `bld_edges`). For a nonempty set E of edges and a set T meeting every edge of E, the function "some edge of E has both ends true" is built with |E| + |T| - 1 gates. Induction on T (Lean: `Finset.induction_on`). If T is empty, a nonempty E cannot have every edge meeting T, so there is nothing to prove. Otherwise T = T0 plus one vertex t not in T0. Split E into E1, the edges containing t, and E2, the rest; |E1| + |E2| = |E|, and every edge of E2 meets T0. The star. Let S be the set of v with tv in E1. The map v -> tv is injective, so |S| <= |E1|. If E1 is nonempty then S is nonempty, and x_t AND (OR of x_v, v in S) is built with 1 + (|S| - 1) = |S| <= |E1| gates (Steps 1, 2, 3); it is true exactly when some edge of E1 has both ends true. The split. "Some edge of E has both ends true" is the OR of the same statement for E1 and for E2 (Lean: `key`). Three cases:

- E1 and E2 nonempty: star OR (the E2 function, built by induction with |E2| + |T0| - 1 gates), with |E1| + (|E2| + |T0| - 1) + 1 = |E| + |T0| = |E| + |T| - 1 gates. The first equality needs |E2|

>= 1, which is this case's hypothesis, so the natural-number minus one does not truncate (Lean: `card_pos` in the `omega` step of `bld_edges`).

- E2 empty: the star alone, |E1| = |E| <= |E| + |T| - 1 gates.
- E1 empty: then E2 = E is nonempty, and the induction hypothesis gives |E| + |T0| - 1 <= |E| + |T| - 1 gates. So a cover vertex with no edge costs nothing.

A person would fix an order of T and give each edge to one of its ends in T; the Lean instead peels one vertex t off T and gives t every edge through it, so each edge goes to the end peeled earliest. The circuit is the same kind: an OR of stars x_t AND (OR of the other ends).

Step 5 (Lean: `bld_hasCircuitSize` for 2.1, `bld_circuit` for 2.2). A function built with s gates has a circuit of size at most s: take the offset m = n and the input string as the wire list. The gate condition `Ok` is the condition `FanInTwo`; for 2.2 the extra invariant gives "no gate is NOT".

Step 6 (Lean: `edge_upper`, `edge_upper_mono`). Apply Step 4 to E = the edge set of G (nonempty, as G has an edge) and the given T (every edge, as an unordered pair, has an end in T), then Step 5. The built function, written with unordered pairs, equals the row's `exists i j, G.Adj i j and ...`; this translation is the last part of the proof. For 2.2, size(D) <= |E| + |T| - 1 with |E| >= 1 gives size(D) + 1 <= |E| + |T|.

Why the circuit has no NOT gate: it is assembled only from input wires (Step 1, no gate) and from gates appended in Step 2, each of which is AND or OR. Why it computes f_G: the star identity in Step 4 and the split identity `key`, each proved in `bld_edges`, and the translation in Step 6.

3. Exact values on paths and cycles

3.1 Tools on circuits

These are the tools of sections 3, 4 and 5; sections 4 and 5 refer to them here.

Lemma 3.1 (counting; Lean: `PermInputs.dep_lower_set` in `H/PermInputs.lean`; the form used in this section is `dep_lower` in `H/Support.lean`). If f on n inputs depends on every input (for each i some x with f(x with x_i flipped) != f(x)), then `HasCircuitSize n f s` implies n <= s + 1. More generally (the set form `PermInputs.dep_lower_set`, used in sections 4 and 5): if f depends on every input of a set T, then every fan-in-two circuit for f with at most s gates has |T| <= s + 1. Proof, as the Lean runs it. Give every wire a SUPPORT: input i has support {i}; a gate has the union of the supports of the wires it names (Lean: `inputSupps`, `gateSupp`, `suppsFrom`). Two inputs that agree on the support of a wire give that wire the same value (Lean: `agree_wiresFrom`). So the output support contains every input f depends on (all n, resp. all of T). Counting: after t gates, any set W of wires has a joint support of at most t + |W| inputs (Lean: `Cnt`, `cnt_inputs`, `cnt_step`, `cnt_suppsFrom`): for t = 0 each wire is one input; adding a gate, if W contains the youngest wire, replace it by the at most two wires it names, which raises |W| by at most 1. At the end, W = {output} gives n <= s + 1 (resp. |T| <= s + 1).

Lemma 3.2 (Lean: `PermInputs.relabel_size`). If h on n inputs has a circuit with at most s gates and pi : {0, ..., n-1} -> {0, ..., m-1} is any map, then x -> h(x o pi) on m inputs has one too. Proof: rename input wire w to pi(w) and gate wire w to w - n + m (Lean: `wmap`, `relabel`); a wire that does not exist stays one that does not exist, so every wire keeps its value (Lean: `rel_fold`, `rel_init`).

The cleaning pass (Lean: `gate_elim` in `H/Elim.lean`; `gate_elim_two`, `two_saved` in `H/Parity.lean`). Fix one input to a constant b. Before the pass every NOT gate is cut down to read only its head wire (Lean: `normGate`; NOT reads its head input in the root semantics, so nothing changes). Then one pass runs through the gates in order (Lean: `elim`, `elimStep`, `simplify`) and gives every old wire a status: a constant, or an alias of a wire of the smaller circuit (Lean: `Stat`). The fixed input has status "constant b"; the other inputs are aliases of themselves. An AND or OR gate is emitted only when both its entries have wire (non-constant) status, and a NOT gate only when its head has wire status (Lean: `simpAnd`, `simpOr`, `simpNot`). In particular a gate that reads a wire of constant status emits no gate (Lean: `simplify_count`, `elimStep_none`): AND or OR with one constant operand becomes a constant or an alias of its other operand, and NOT of a constant is a constant. The pass computes the restriction (Lean: `inv_elim`), and it never emits more gates than it reads (Lean: `elim_len`). The number of gates it saves is counted by Lemma 5.7.10.

Lemma 3.3 (one- and two-gate elimination). Let f be on n + 1 inputs, depending on input p. (a) (Lean: `Forest.one_gate_at`, `H/ForestElim.lean`, over `gate_elim`.) If f with x_p := b is not constant, then a circuit for f with at most s + 1 gates gives one for f with x_p := b with at most s gates. (b) (two-gate elimination; Lean: `PermInputs.gate_elim_at`, from `gate_elim_two` in `H/Parity.lean`, whose core is `two_saved`.) If f restricted to x_p = b is non-constant for both b and f has a circuit with s gates, then for some b the function x -> f(x with x_p := b), on the same n + 1 inputs, has a circuit with at most s - 2 gates. Proof, as the Lean runs it. Swap p with the last input (Lemma 3.2 with a transposition; Lean: `snoc_swap`), so it suffices to fix the last input, and run the cleaning pass with the last input fixed to b. The output never gets a constant status, since the restriction is not constant; in particular the output is not the fixed input wire, whose status is constant (Lean: `hout` in `gate_elim`, `hon` in `two_saved`). Some gate g names the fixed input, or else the output support misses that input and f would not depend on it (Lean: `supp_noread`). (a) The gate g reads a constant and emits nothing (Lean: `elim_len_lt`). (b) In `two_saved`: if a second gate also names the fixed input, take b = false: two gates are not emitted. Otherwise take b = 1 if g is OR and b = 0 if g is AND or NOT (Lean: `fixBit`; for NOT any value does, and the Lean takes 0); then g gets a constant status (Lean: `simplify_const`, `elimStep_const`). If a later gate names g, it is not emitted either: two gates saved. If no gate names g, then g is not the output (its status is constant) and no wire other than the fixed input and g carries it in its support (Lean: `supp_noreadD`), so the output support misses the input, against dependence. Last, rename back (Lemma 3.2) to keep the arity n + 1: the restricted function stays a function of all n + 1 inputs that ignores input p. A person would fix p in place.

3.2 Paths

Theorem 3.4. For all natural numbers n and s, P_n(x) = OR over i + 1 = j of (x_i AND x_j) on n inputs has a fan-in-two AND, OR, NOT circuit with at most s gates if and only if n + floor(n/2) <= s + 2. So its circuit size is n + floor(n/2) - 2 for n >= 2, and 0 for n <= 1. Lean: `lean/PEqualNP/MetaComplexity/L102.lean`, `PEqualNP.MetaComplexity.L102`; checked as L-meta-102, run-011301.

The proof is `path_size` (`H/PathSizeUpper.lean`): the two directions `path_lower` and `path_upper`.

Lower bound (Lean: `path_lower`, `H/PathSizeLower.lean`). The Lean proves a stronger statement on a class with free inputs. The class (Lean: `G N k`). On N inputs, G_{N,k}(x) = (x_0 OR ... OR x_{k-1}) OR (OR over i >= k of x_i AND x_{i+1}): k free inputs, then a path on the d = N - k inputs k, ..., N -

1. P_n = G_{n,0} (Lean: `path_eq_G`). Claim (Lean: `G_lower`). If d >= 2 and G_{N,k} has a circuit with s gates, then N + floor(d/2) <= s + 2. Strong induction on d.

- d = 2 or 3. G_{N,k} depends on every input (Lean: `G_dep`: a free input flips the all-false point; a path vertex p flips the point whose only true input is a path neighbour of p). Lemma 3.1 gives N <= s + 1, and floor(d/2) = 1.
- d >= 4. Let p = k, the end of the path next to the free inputs. G_{N,k} depends on p (`G_dep`), and for either value b at p the restriction is not constant: it is 0 at the point with every other input false and 1 at the point with x_{k+2} = x_{k+3} = 1 (this uses d >= 4). Lemma 3.3(b) gives b and a circuit with s' <= s - 2 gates for the restriction. Rename the inputs by the map that skips index k (Lean: `del`, with Lemma 3.2) to get a function on N - 1 inputs. b = 0 (Lean: `G_del_false`): the result is G_{N-1,k}, with d - 1 path vertices. By induction, (N - 1) + floor((d-1)/2) <= s' + 2 <= s, and floor(d/2) <= floor((d-1)/2) + 1, so N + floor(d/2) <= s + 2. b = 1 (Lean: `G_del_true`): the edge k(k+1) becomes the single input x_{k+1}, which is now a free input, and the result is G_{N-1,k+1}, with d - 2 >= 2 path vertices. By induction, (N - 1) + floor(d/2) - 1 <= s' + 2 <= s, so N + floor(d/2) <= s + 2.

Then `path_lower`: for n <= 1 the inequality n + floor(n/2) <= s + 2 holds for every s; for n >= 2 apply the claim with k = 0, d = N = n. A person might take the lower bound from the forest bound (L-meta-104) at the path; the Lean gives the path its own elimination, with the free inputs kept at the low indices so that the renaming `del` preserves the form of the class.

Upper bound (Lean: `path_upper`, `pathGates`, `three_gates`, `pathB_step`). For n <= 1 the empty gate list with output index n works: that wire does not exist and reads false, and P_n is constant false. For n >= 2 a gate list built at any offset m >= n (Lean: `pathGates`, with `pathB` the path function read off a wire list) computes P_n with exactly n + floor(n/2) - 2 gates, by induction in steps of two:

- n = 2: one gate AND(x_0, x_1).
- n = 3: two gates OR(x_0, x_2), then AND(x_1, that).
- from n to n + 2 (n >= 2): append three gates OR(x_{n-1}, x_{n+1}), AND(x_n, that), OR(previous output, that). Correct by the identity (Lean: `pathB_step`) P_{n+2}(x) = P_n(x) OR (x_n AND (x_{n-1} OR x_{n+1})), checked on the 16 values of the four quantities involved; the gate count rises by 3 while n + floor(n/2) rises by 3.

`pathB_ofFn` identifies `pathB` on the input string with the row's function. This circuit is built directly, not through Theorem 2.1.

3.3 Cycles

The cycle proof calls another checked row, which this section does not prove.

Theorem 3.5 (reader lemma). Let f be on n + 1 inputs. Assume: f depends on some input; for every input p that f depends on and each b, f restricted to x_p = b is not constant; and there are no two distinct inputs p, q that f depends on and values a, b with, for every x, f(x_p = not a, x_q = b) = f(x_p = a, x_q = not b) = f(x_p = not a, x_q = not b) (no PAIR FUNNEL: three of the four restrictions to (x_p, x_q) coincide). If f has a circuit with s gates, then for some input y either f with x_y := 0 has a circuit with at most s - 3 gates, or f with x_y := 1 has one with at most s - 4 gates (arity kept). Lean: `lean/PEqualNP/MetaComplexity/L107.lean`, `PEqualNP.MetaComplexity.L107` (proof term

`PEqualNP.MetaComplexity.Helpers.Readers.fanout_step_3_4`, `H/ReadersExport.lean`); checked as L-meta-107, run-011915 (the run named in the table of section 0).

Theorem 3.6. For all natural numbers n >= 5 and s, C_n(x) = OR over (i + 1) mod n = j of (x_i AND x_j) on n inputs has a fan-in-two AND, OR, NOT circuit with at most s gates if and only if n + ceil(n/2) <= s + 1 (written n + (n + 1) / 2 <= s + 1). So its circuit size is n + ceil(n/2) - 1. Lean: `lean/PEqualNP/MetaComplexity/L106.lean`, `PEqualNP.MetaComplexity.L106`; checked as L-meta-106, run-012105.

The proof term is `cyc_exact_of_readers L107` (`H/CycleCore.lean`), which is `cyc_exact_of_core (cyc_core_of_readers L107)`. In the Lean, C_n is `cyc n` (`H/CycleUpper.lean`).

Step 1, the core (Lean: `StmtCycCore`, `cyc_core_of_readers`). If n >= 5 and C_n has a circuit with s gates, then for some input v: s >= 3 and C_n with x_v := 0 has a circuit with s - 3 gates, or s >= 4 and C_n with x_v := 1 has one with s - 4 gates. (`StmtCycCore` also assumes s is below the claimed size; the proof does not use that hypothesis.) Proof: Theorem 3.5 at f = C_n, after checking its three hypotheses. (i) C_n depends on every input p (Lean: `cyc_dep`, n >= 3): at the point whose only true input is p + 1 (mod n), C_n is 0, and setting x_p = 1 makes it 1. (ii) No restriction is constant (Lean: `cyc_nonconst`, n >= 3): with x_p = b, C_n is 0 when every other input is 0 (at most one input is true) and 1 when x_{p+1} = x_{p+2} = 1. (iii) No pair funnel (Lean: `cyc_no_funnel`, from `exists_nbr` and `cyc_sens`). For p != q there is a neighbour r of p with r != q and r not adjacent to q (Lean: `exists_nbr`: if q is p + 1 or p + 2, take r = p - 1, otherwise r = p + 1; this is the one place in Step 1 where n >= 5 is used; Step 2 uses it again at value 1, where d = n - 3 >= 2). At the point whose only true input is r, with x_q fixed to any c: x_p = 1 gives C_n = 1 (edge pr), and x_p = 0 gives C_n = 0, because the true inputs lie in {r, q}, which is not an edge (Lean: `cyc_sens`). Taking c = not b, the funnel's second equality f(x_p = a, x_q = not b) = f(x_p = not a, x_q = not b) fails. The Lean refutes only this equality, not the whole three-way coincidence. Theorem 3.5 then returns y and s' with s' + 3 <= s (resp. s' + 4 <= s), and a circuit of s' gates is one of s - 3 (resp. s - 4) gates (Lean: `size_mono`).

Step 2, the lower bound (Lean: `cyc_exact_of_core`, `H/CycleUpper.lean`). Suppose C_n (n >= 5) has a circuit with s gates and s + 1 < n + ceil(n/2). Step 1 gives v. Moving v to the end (Lean: `restrict_last`, with the rotation `rot`, `cyc_rot` and the renaming `piv`). C_n is invariant under rotating the inputs; rotating v to position n - 1 and then dropping the fixed input (Lemma 3.2) gives a circuit of the same size, on n - 1 inputs, for y -> C_n(y, b) (the last input fixed to b). A person would fix v directly by symmetry; the Lean does it with an explicit rotation and renaming. Value 0 (Lean: `cyc_snoc_false`, `cyc_false_lower`). C_n with the last input 0 is P_{n-1}, so by the path lower bound (`path_lower`), (n - 1) + floor((n-1)/2) <= (s - 3) + 2. But s <= n + ceil(n/2) - 2 gives (s - 3) + 2 <= n + ceil(n/2) - 3, and (n - 1) + floor((n-1)/2) = n + ceil(n/2) - 2 (as floor((n-1)/2) = ceil(n/2) - 1). Contradiction (Lean: `arith_false`). Value 1 (Lean: `cyc_snoc_true`, `cyc_true_lower`, with the renaming `tau`: 0 -> 0, n - 2 -> 1, j -> j + 1 otherwise). With the last input 1, its two neighbours 0 and n - 2 become free inputs (each edge to the fixed vertex reduces to the neighbour's input), and the remaining inputs 1, ..., n - 3 form a path. After the renaming this is G_{n-1,2}: two free inputs and a path on d = n - 3 >= 2 inputs. By the claim of section 3.2 (`G_lower`) and Lemma 3.2, (n - 1) + floor((n-3)/2) <= (s - 4) + 2. But (s - 4) + 2 <= n + ceil(n/2) - 4, and (n - 1) + floor((n-3)/2) = n + ceil(n/2) - 3 (as floor((n-3)/2) = ceil(n/2) - 2). Contradiction (Lean: `arith_true`). So a circuit with s gates forces n + ceil(n/2) <= s + 1.

Step 3, the upper bound (Lean: `cyc_upper`, `cyc_eq_path_or`; n >= 3 suffices). For n = m + 1, C_n(x) = P_m(x_0, ..., x_{m-1}) OR (x_m AND (x_{m-1} OR x_0)) (Lean: `cyc_eq_path_or`). Take the path gates for P_m on the inputs 0, ..., m - 1 (m + floor(m/2) - 2 gates) and append OR(x_{m-1}, x_0),

AND(x_m, that), OR(path output, that). The total is m + floor(m/2) + 1 = (n - 1) + floor((n-1)/2) + 1 = n + ceil(n/2) - 1 gates. The path gates here are `pathGates` of `H/CyclePathGates.lean` (namespace `CycleSize.PG`), a verbatim copy of the one in `H/PathSizeUpper.lean`, kept separate only to avoid a name clash between imported files. As for the path, this circuit is built directly, not through Theorem 2.1.

A note on the files: the module comment of `H/CycleUpper.lean` still calls `StmtCycCore` "the gate-elimination core that is still open"; that comment predates `H/CycleCore.lean`, where the core is derived from L-meta-107. The comment is not part of any statement.

4. Forests (L-meta-104): gate elimination at a good leaf

4.0 Statement

Theorem 4.1 (L-meta-104). Let n, k, s be natural numbers and let G be a simple graph on the vertex set {0, ..., n-1}. Assume

1. G is acyclic;
2. G has at least one edge;
3. every vertex cover of G (every set T of vertices such that each edge has an end in T) has at least k vertices;
4. the function f_G(x) = 1 iff there are i, j with ij an edge of G and x_i = x_j = 1 has a fan-in-two circuit with at most s gates.

Then |E| + k <= s + 1.

Lean: `lean/PEqualNP/MetaComplexity/L104.lean`, `PEqualNP.MetaComplexity.L104`; checked as L-meta-104, run-011343.

The circuit model is the root one of 1.1 to 1.3 (`FanInTwo`, `HasCircuitSize`, `Circuit.size`): constant gates and repeated wires are allowed, every gate counts, inputs do not. Taking k equal to the vertex cover number tau(G), the theorem says every such circuit has at least |E| + tau(G) - 1 gates.

4.1 The function with free inputs, and the claim proved by induction

For a set A of vertices, F_{G,A} is the function of 1.5 (Lean: `Forest.F`, `H/ForestBound.lean`). Inputs in A are "free": they enter as single OR terms.

Claim Q (Lean: `Forest.claimQ`). Let m, n, k, s be natural numbers, G a simple graph on {0, ..., n-1} with exactly m edges, A a set of vertices. Assume G is acyclic; no edge has an end in A; every vertex cover of G has at least k vertices; A is nonempty or G has an edge; and F_{G,A} has a fan-in-two circuit with at most s gates. Then |A| + m + k <= s + 1.

Theorem 4.1 from Claim Q (Lean: `Forest.forest_bound`). The function of the theorem is F_{G, empty set} (the two agree at every x), and A empty meets the hypotheses on A. Claim Q gives 0 + |E| + k <= s + 1. (The Lean counts edges once as `edgeFinset.card`, in the row, and once as `edgeSet.ncard`, in Claim Q, and identifies the two.)

4.2 Tools

The counting tool and the one- and two-gate eliminations are those of section 3.1: Lemma 3.1 in its set form (`PermInputs.dep_lower_set`), Lemma 3.3(a), one gate at a chosen value

(`Forest.one_gate_at`), and Lemma 3.3(b), two gates at some value (`PermInputs.gate_elim_at`), both run through the cleaning pass of section 3.1. The one tool special to forests is the following.

Tool C, restrictions of F at a leaf (Lean: `Forest.restrict_false`, `Forest.restrict_true`, `H/ForestBound.lean`). Let l be a leaf with neighbour p, l not in A.

- x_l := 0 gives F_{G - lp, A}, where G - lp deletes the one edge lp (`restrict_false`, which uses that every neighbour of l is p).
- If moreover every neighbour of p is l or q, then x_l := 1 gives F_{G1, A + p} with G1 = G - {pl, pq} (`restrict_true`): the edge lp becomes the single term x_p, which joins the free inputs, and the edge pq (if present) is absorbed by x_p.

Edge counts (`ncard_del1`, `ncard_del2`): G - lp has m - 1 edges; G1 has fewer than m edges and at least m - 2. Cover facts (`cov_insert`, `cov_swap`):

- if every vertex cover of G has at least k vertices and the deleted edges all contain p, then every vertex cover of the smaller graph has at least k - 1 vertices (add p to it);
- if p carries a second leaf l' (l' not l), then every vertex cover T of G - lp still has at least k vertices: if p is in T, T covers G; if not, l' is in T (to cover pl'), and (T minus l') plus p covers G and has at most |T| vertices.

Dependence (`Forest.F_dep`, `F_dep_A`): F_{G,A} depends on each input of A, and on each end of an edge whose other end is not in A (set the other end to 1, everything else to 0).

4.3 A good leaf exists

Lemma 4.2 (Lean: `Forest.forest_good_leaf`, `H/ForestGraph.lean`). A finite acyclic graph with an edge has a leaf l with neighbour p such that either (a) every neighbour of p is l or one vertex q, or (b) p has a second leaf neighbour l' different from l.

Proof, as in the Lean. A longest path exists (`exists_longest_path`: path lengths are bounded by the number of vertices). Take one, l, p, ..., z; it has length at least 1 since G has an edge. Every neighbour w of l is p: if w lies on the path, acyclicity forces w to be the vertex after l (Mathlib `IsAcyclic.eq_snd_of_adj_start`); if not, w, l, p, ..., z is a longer path. A neighbour w of p off the path is a leaf: a neighbour u of w other than p would either close a cycle or extend w, p, ... to a path longer than the longest. If such an off-path neighbour other than l exists, (b) holds. Otherwise every neighbour of p other than l lies on the rest of the path p, ..., z and so, by acyclicity again, is the vertex after p; call it q, and (a) holds.

Case (a) includes p having l as its only neighbour (an isolated edge lp).

4.4 The induction

Proof of Claim Q (Lean: `Forest.claimQ`). Strong induction on m, for all n, G, A, k, s at once.

No edge. Then the empty set is a vertex cover, so k = 0, and m = 0 (`noedge_ncard`). F_{G,A} depends on every input of A (`F_dep_A`), so Lemma 3.1 gives |A| <= s + 1, which is the claim.

An edge. Take l, p as in Lemma 4.2. Neither l nor p is in A (no edge meets A). F depends on l and on p (`F_dep`), so Lemma 3.1 applied to {l, p} gives 2 <= s + 1. Put G0 = G - lp; it is acyclic, has m - 1 edges, and no edge of it meets A; by Tool C, x_l := 0 turns F_{G,A} into F_{G0,A}.

Case (a): every neighbour of p is l or q.

- *(a1) A is nonempty or G0 has an edge.* Both restrictions at l are non-constant: at 0 the function is F_{G0,A} with A or an edge present (`F_nc`: it is 0 at all-zero and 1 at all-one); at 1 it is F_{G1, A + p} with A + p nonempty. Lemma 3.3(b) (`gate_elim_at`) gives b and s' with s' + 2 <= s and a circuit of at most s' gates for the restriction.
- b = 0: the residual is F_{G0,A}. Vertex covers of G0 have at least k - 1 vertices (`cov_insert`). Induction: |A| + (m - 1) + (k - 1) <= s' + 1 <= s - 1, hence |A| + m + k <= s + 1. The Lean uses k - 1 here even when a leaf's deletion leaves the cover number unchanged; the two gates pay for it.
- b = 1: the residual is F_{G1, A + p}; G1 is acyclic and no edge of G1 meets A + p (`del_p_adj`: no edge of G1 is at p). Covers of G1 have at least k - 1 vertices (`cov_insert`), |A + p| = |A| + 1 (p not in A), and m <= |E(G1)| + 2 with |E(G1)| < m. Induction: (|A| + 1) + |E(G1)| + (k - 1) <= s' + 1 <= s - 1, hence |A| + m + k <= |A| + |E(G1)| + 2 + k <= s + 1.
- *(a2) A is empty and G0 has no edge.* Then G is the single edge lp, m = 1, and {p} is a vertex cover, so k <= 1. From 2 <= s + 1 above, k <= s, i.e. 0 + 1 + k <= s + 1. No elimination is used in this case: the Lean closes it by Lemma 3.1 at {l, p}. (A person might fix x_l := 1 and land on the single free input x_p; the Lean does not.)

Case (b): p has a second leaf l'. From 2 <= s + 1 write s = s0 + 1. The edge pl' survives in G0, so F with x_l := 0, which is F_{G0,A}, is non-constant. Lemma 3.3(a) at the value 0 (`one_gate_at`) gives a circuit with at most s0 gates for F_{G0,A}. Every vertex cover of G0 still has at least k vertices (`cov_swap`). Induction: |A| + (m - 1) + k <= s0 + 1 = s, hence |A| + m + k <= s + 1.

When Lemma 4.2 returns both (a) and (b), the Lean returns (b) (it tests for a second leaf before choosing q); the induction handles either. This ends the proof of Claim Q and, with section 4.1, of Theorem 4.1.

In potential terms, with Phi = |A| + |E| + k - 1 and for k at least 1 (at k = 0 the natural-number k - 1 truncates): the step at a good leaf lowers Phi by 2 at x_l := 0 in case (a) (one edge, and k to k - 1), by at most 2 at x_l := 1 in case (a) (at most two edges and k to k - 1, A up by one), and by 1 in case (b) (one edge, k kept); the circuit loses at least 2, 2 and 1 gates respectively.

5. Graphs with long threads (L-meta-204, L-meta-115, L-meta-113)

The three theorems of this section are statements our checker accepted. As for the whole note, this prose is checked by nothing: it is a reading of those proofs, to be compared line by line with the Lean. Two internal working notes, not in the public repository, are mentioned below as the working note; they are on paper and are cited only to say where the Lean takes a different route.

5.0 The three statements

Circuits, size, f_G and F_{G,A} are those of section 1 (1.1 to 1.5), and a HUB is as in section 0: a vertex with at least three neighbours.

Theorem 5.0.1 (L-meta-204). Let n, k, s be natural numbers, G a simple graph on {0, ..., n-1} and A a set of vertices. Assume: (i) every cycle of G has length at least 7; (ii) for any two distinct hubs u, v, every walk from u to v has length at least 7; (iii) no vertex of A is an end of an edge; (iv) A is nonempty or G has an edge; (v) every vertex cover T of G has |T| >= k. If F_{G,A} has a circuit with at most s gates, then |A| + |E| + k <= s + 1. Lean: `lean/PEqualNP/MetaComplexity/L204.lean`, `PEqualNP.MetaComplexity.L204` (proof term `PEqualNP.MetaComplexity.EdgeFn.stmtLB_final`); checked as L-meta-204, run-013633.

Theorem 5.0.2 (L-meta-115). Let G be a simple graph on {0, ..., n-1} in which every cycle has length at least 9 and every walk between two distinct vertices of degree at least 3 has length at least 7. For every circuit D with no NOT gate (fan-in two, AND and OR only) that computes f_G(x) = OR over edges ij of x_i AND x_j, there is a vertex cover T of G with |E| + |T| <= size(D) + 1. Lean: `lean/PEqualNP/MetaComplexity/L115.lean`, `PEqualNP.MetaComplexity.L115` (proof term `PEqualNP.MetaComplexity.Loc.Final.row`); checked as L-meta-115, run-012892.

Theorem 5.0.3 (L-meta-113). Let G be a simple graph on {0, ..., n-1} in which every cycle has length at least 9 and every walk between two distinct vertices of degree at least 3 has length at least 9. For every fan-in-two circuit C computing f_G there is a fan-in-two circuit D with no NOT gate, computing the same function as C, with size(D) <= size(C). Lean: `lean/PEqualNP/MetaComplexity/L113.lean`, `PEqualNP.MetaComplexity.L113` (proof written in the file); checked as L-meta-113, run-014387.

Order of this section: 5.1 to 5.8 prove 5.0.1 in the order its proof term runs (`stmtLB_final` := `lb_of_front stmtFront`, which unfolds to `lb_of_ml (ml_of_front stmtFront)`, then `lb_of_pieces stmtG3 stmtG4`, then `lb_of_a1 (a1_of_pieces ..)`, `H/EdgeFnFinal.lean`, `H/EdgeFnOfFront.lean`, `H/EdgeFnGraphPlan.lean`): from the row down to the induction, then the two graph steps, then the circuit lemma and its pieces. 5.9 proves 5.0.3 from 5.0.1; 5.10 cites the separate proof of 5.0.2.

5.1 Objects used by the proof of 5.0.1

The working model is least clean gate lists (`CleanL`, `vl`, `H/Parity3.lean`, `H/Parity4.lean`; `Readers.Least`, `H/Readers.lean`): every gate is NOT of one earlier wire or AND/OR of two earlier wires, the list computes the function at a named output wire, and no clean list for the function is shorter. Gate number p of a list on N inputs is wire N + p. A gate READS a wire if the wire is among its operands (`Readers.Reads`). Transfer to the root circuits of 1.3: a circuit for a non-constant function can be cleaned without growing (`clean_of`), so a non-constant function with a circuit of size s has a least clean list of length <= s (`Readers.exists_least`); a clean list is a circuit of its length (`Readers.has_of_clean`). For a non-constant f a least clean list is reduced: no gate computes a constant (`Least.no_const`), no gate computes the function of an earlier wire, inputs included (`Least.no_alias`), every gate is the output or is read by a later gate (`Least.live`, all three in `H/Readers.lean`), and no gate reads both a wire a and a gate that reads a (`Least.no_tri`, `H/ReadersPat.lean`).

- Potential (as in 1.6 and 1.7). tau(G) is the least size of a vertex cover (`Loc.tau`, `H/LocGraph.lean`; attained, `tau_attained`). Phi1(G, A) = |A| + |E| + tau(G) (`Loc.Phi1`). The bound at (G, A) is: every s with a circuit of size s for F_{G,A} has Phi1(G, A) <= s + 1 (`Loc.BoundOf`). `AFree G A`: no edge has an end in A. `delV G X`: G with every edge at a vertex of X deleted, the number of inputs kept (`Loc.delV`).
- Class (`EdgeFn.T7`, `H/EdgeFnGraphPlan.lean`): hypotheses (i) and (ii) of 5.0.1.
- Good leaf (`EdgeFn.GoodLeaf`): a leaf l at p such that p has at most one neighbour besides l, or p has a second leaf l'.
- Run (`Loc.Run`, `H/LocGraph.lean`): six distinct vertices a', a, s, t, b, b' such that the neighbours of a are exactly a', s; of s exactly a, t; of t exactly s, b; of b exactly t, b'; and a', b' each have at most one neighbour besides a, resp. b. Its four inner vertices D = {a, s, t, b} are the deep vertices.
- Deep vertex (`Loc.Deep`): v has exactly two neighbours u1 != u2, each with at most one neighbour besides v, and some edge avoids v. Every inner vertex of a run is deep (`Run.deep_a`, `deep_s`, `deep_t`, `deep_b`; `deep_mem`).

- For a function F on N inputs and a set D of inputs, `DeepHyp F D` (`H/EdgeFnPlan.lean`) is the conjunction of: H1 for y in D and b in {0, 1}, the restriction F|x_y=b is not constant; H2 no two distinct y, y' in D form a pair funnel (`Readers.PairFunnel`: of the four restrictions of F to the values of (x_y, x_y'), three coincide); H3 (window) for y in D and every x with F(x with D switched off) = 0, there is x' equal to x outside D, with x'_y = 0, F(x') = 0 and F(x' with x_y := 1) = 1; H4 if F(x with D switched off) = 1 then F(x) = 1; H5 F(all zero) = 0.
- A LITERAL of y is the input wire y or a gate NOT y (`IsLit`). A wire is D-FREE if no input of D is in its support (`DFree`). `OneReader D gs`: every input of D, and every gate NOT y with y in D, has at most one reader. A FRONTIER gate (`Front`): an AND or OR gate whose operands are a literal l of some y in D and a D-free wire h (h is its PARTNER).
- Tail (`TailOK`, `TailGE`): a gate is tail-OK if it is a NOT gate, or an OR gate whose value is below F at every input; a list has tail at least t if its last t gates are tail-OK.
- Redirect-and-delete (`rdel N p a gs`): delete gate p and let every later read of wire N + p read the earlier wire a (later wires renumbered by `rd`).
- `Drop F y b k s`: F|x_y=b has a circuit of size s' with s' + k <= s. `Pays F y s`: drops (3, 3), (4, 2) or (2, 4) at the values (0, 1) (`H/Readers.lean`).

The elimination count used everywhere below is `Readers.save` (`H/Readers.lean`): fix input i of a clean list to b; if the restriction is not constant, then for every set S of positions each of which reads a wire whose status after propagating the constant is a constant (`RC`), the restriction has a circuit of size (length) - |S|. Its proof runs the cleaning pass of section 3.1 (`passG`) and counts with `elim_count` (Lemma 5.7.10); a person would say: every gate reading a constant is removed or becomes a copy, and the count is the number of such gates.

5.2 From the class bound to the row (`lb_of_a1`)

Given the class form (5.3) at (G, A): Phi1(G, A) <= s + 1. A least cover T exists with |T| = tau(G) (`Loc.tau_attained`) and hypothesis (v) gives k <= |T| = tau(G); so |A| + |E| + k <= Phi1(G, A) <= s + 1. (Lean: `EdgeFn.lb_of_a1`.) Hypothesis (iv) of 5.0.1 is not used by the Lean proof: if A is empty and G has no edge, the base case of 5.3 gives 0 <= s + 1 directly.

5.3 The induction (`a1_of_pieces`)

Theorem 5.3.1 (class form). Let P be a class of graphs closed under `delV` (`Hered`), in which every graph with an edge has a good leaf or a run (`Dich`). Then for every G in P and every A with `AFree G A`, the bound holds at (G, A). (Lean: `EdgeFn.a1_of_pieces`; it takes as hypotheses the graph pieces G3, G6, the circuit lemma `StmtML` and the tools `ToolRes0`, `ToolRes1`, `ToolDrop0`, `ToolDrop1`, all proved below.)

Proof. Strong induction on the number of edges, with A carried and no induction on |A|. The induction hypothesis at G (`IHV G`) is: the bound at (delV G X, A') for every X and every A' with `AFree (delV G X) A'` and fewer edges than G. It follows from the outer hypothesis because delV G X is again in P.

Base: G has no edge. F_{G,A} depends on every a in A (Lean: `Forest.F_dep_A`), so by the counting Lemma 3.1 every circuit of size s has |A| <= s + 1 (Lean: `PermInputs.dep_lower_set`); |E| = 0 (Lean: `Forest.noedge_ncard`) and tau(G) = 0 (the empty set covers). So Phi1(G, A) <= s + 1.

Step: G has an edge. By `Dich`, G has a good leaf, and the leaf step 5.5 applies (Lean: `stmtG6`), or a run, and the deep step 5.6 applies (Lean: `deep_step`). QED.

For the class T7 the two hypotheses are 5.4.1 and 5.4.2 (Lean: `lb_of_pieces` applies `a1_of_pieces stmtG1 (dich_of_g4 stmtG4) stmtG3 stmtG6 ..`).

5.4 The class T7: heredity and the dichotomy

Lemma 5.4.1 (heredity; `stmtG1`). If G is in T7, so is delV G X. Proof. delV G X is a subgraph of G (`delV_le`). A cycle of it maps to a cycle of G of the same length (`Walk.mapLe`); its hubs are hubs of G, since neighbour sets only shrink; a walk between them maps to a walk of G of the same length. QED.

Lemma 5.4.2 (dichotomy; `dich_of_g4` with `stmtG4`). A graph G in T7 with an edge has a good leaf or a run. Proof. `dich_of_g4`: if some leaf l at p has p with at most one further neighbour, or p has two leaves, that is a good leaf. Otherwise `stmtG4` produces a run, as follows.

Thread lemma (`thread7`). Assume no leaf l at p has p with at most one further neighbour. Let u be a vertex and v a neighbour of u with exactly two neighbours, and assume u is a hub or G has no hub. A THREAD at u is a path x, ..., u of length at least 1 whose vertices other than u have exactly two neighbours each (`Loc.Struct.Good`). Then some thread at u has length at least 6. Proof. The edge v u is a thread; take a longest one, q from x to u (`Loc.Struct.exists_max`). Suppose its length is at most 5. The vertex x has exactly two neighbours; one is the next vertex of q, call the other z (`two_other`). Either z = u, or `classify` puts z in one of its three cases: exactly two neighbours, at least three, or at most one. (a) z = u. The edge u x is not an edge of q (Lean: `hne`): the only edge of the path q at its first vertex x goes to the next vertex of q, and z = u is not that vertex. So the edge u x and q close a cycle of length (length of q) + 1 <= 6, against (i). (b) z has exactly two neighbours. If z lies on q, it is not at position 0 (z is adjacent to x), not at position 1 (z is the neighbour of x off q) and not at the end (z != u, case (a)); so it has its two neighbours on q and the neighbour x, three distinct neighbours, impossible (`two_three`). If z is not on q, then z, x, ..., u is a longer thread, against the choice of q. (c) z is a hub. If u is a hub, z, x, ..., u is a walk of length at most 6 between distinct hubs, against (ii). If G has no hub, this case does not occur. (d) z has at most one neighbour, so z is a leaf at x, and x has exactly the neighbours z and the next vertex of q: a leaf at a vertex with one further neighbour, excluded by assumption. So q has length at least 6. QED.

From a thread of length at least 6 the vertices at positions 0, ..., 5 form a run (`Loc.Struct.run_of_thread`): they are distinct (q is a path), consecutive ones are adjacent, and each has exactly two neighbours, which for positions 1 to 4 are their two neighbours on q.

Proof of `stmtG4`. Assume G has an edge, no leaf l at p with p of at most one further neighbour, and no two leaves at one vertex. If G has a hub u: every neighbour m of u has exactly two neighbours or is a leaf, since a neighbour that is a hub would give a walk of length 1 between distinct hubs, against (ii). Of three distinct neighbours of u, at most one is a leaf (no two leaves at u), so some neighbour v has exactly two neighbours; the thread lemma at (u, v) gives a run. If G has no hub: take an edge ij. If j has exactly two neighbours, apply the thread lemma at (i, j); else j is a leaf at i; if i has exactly two neighbours apply it at (j, i); else i is a leaf too, and i, j form an isolated edge, a leaf at a vertex with no further neighbour, excluded. QED.

The class T7 is read by the proof of 5.0.1 only in 5.4.1 and 5.4.2: the cycle bound 7 in case (a), the hub distance 7 in case (c) and in `stmtG4`. Every later step reads only the run (5.6) or the good leaf (5.5).

5.5 The leaf step (`stmtG6`)

Fact 5.5.0 (Lean: `hinv1`, proved inline in `stmtG6`). If `AFree G A`, then `AFree (delV G {l, p}) (A + p)`: an edge of delV G {l, p} has no end in {l, p}, and none in A.

Lemma 5.5.1. If `AFree G A`, the induction hypothesis `IHV G` holds, and G has a good leaf, then the bound holds at (G, A). Proof. Let l be a leaf at p, and let a circuit of size s compute F = F_{G,A}. Neither l nor p is in A. F depends on x_l and on x_p (Lean: `Forest.F_dep`), so s + 1 >= 2 (Lean: `PermInputs.dep_lower_set` on {l, p}). Fixing x_l = 0 gives F_{delV G {l}, A} (Lean: `Res.ToolRes0_proof`), and Phi1(G, A) <= Phi1(delV G {l}, A) + 2, with exactly one edge fewer (Lean: `leaf_drop0`: the edge lp goes, and a least cover of the rest plus p covers G, `cov_insert`).

Case 1: p has at most one neighbour q besides l. Then Phi1(G, A) <= Phi1(delV G {l, p}, A + p) + 2 with fewer edges (Lean: `leaf_drop1`: at most the two edges pl, pq go (`ncard_del2`); a cover of the rest plus p covers G; A gains p). 1a. A is nonempty or delV G {l} has an edge. Fixing x_l = 1 gives F_{delV G {l, p}, A + p} (Lean: `Res.ToolRes1_proof`, the neighbours of l being exactly {p}). Both restrictions are not constant (Lean: `Forest.F_nc`; at 1, because A + p is nonempty). Lemma 3.3(b) gives a value b and a circuit of size s' <= s - 2 for F|x_l=b (Lean: `PermInputs.gate_elim_at`). The induction hypothesis at the residual (fewer edges; A meets no edge of delV G {l} by `afree_delV`, and A + p no edge of delV G {l, p} by Fact 5.5.0) gives Phi1(residual) <= s' + 1, and Phi1 dropped by at most 2 at either value. So Phi1(G, A) <= s + 1. 1b. A is empty and delV G {l} has no edge. No elimination is used; the Lean runs three steps. `leaf_drop0` gives Phi1(G, A) <= Phi1(delV G {l}, A) + 2. The residual has no edge, so it has 0 edges (Lean: `Forest.noedge_ncard`) and the empty set covers it, so its tau is 0 (Lean: `Loc.tau_le` at the empty cover); with A empty, Phi1(G, A) <= 2. And 2 <= s + 1 was shown above, from Lemma 3.1 on {l, p}: F depends on both inputs (non-constancy alone would not do, since x_l alone is non-constant and costs no gate). The Lean treats an isolated edge (p with no neighbour besides l) inside Case 1: by Lemma 3.3(b) when A is nonempty or G has another edge (1a), by counting alone when A is empty and lp is the only edge (1b); it uses no one-gate step at x_l = 1.

Case 2: p has a second leaf l'. Fixing x_l = 0: the restriction F_{delV G {l}, A} is not constant (the edge pl' survives). Lemma 3.3(a) needs s >= 1: from 2 <= s + 1, shown above by Lemma 3.1 on {l, p}, write s = s0 + 1 (Lean: the `obtain` of s0 in `stmtG6`); then it gives a circuit of size s0 = s - 1 (Lean: `Forest.one_gate_at`). Every cover of delV G {l} has size at least tau(G) (Lean: `Forest.cov_swap`: a cover T of G - lp contains p or l'; if p, it covers G; if not, T - l' + p covers G and is no larger, since l' has no neighbour but p). So tau(G) <= tau(delV G {l}), exactly one edge goes, A is unchanged: Phi1 drops by at most 1, and the induction hypothesis closes. QED.

5.6 The deep step (`deep_step`)

Fact 5.6.0 (Lean: `hinv1`, proved inline in `deep_step`). If `AFree G A`, then `AFree (delV G {y, u1, u2}) (A + {u1, u2})`: an edge of delV G {y, u1, u2} has no end in {y, u1, u2}, and none in A.

Lemma 5.6.1. If `AFree G A`, `IHV G` holds and G has a run a' a s t b b', then the bound holds at (G, A). Proof. Let a circuit of size m compute F = F_{G,A} (m, not s: s names a vertex of the run); let D = {a, s, t, b}. (1) F and D satisfy `DeepHyp`, and |D| >= 2 (Lean: `stmtG3`, Lemma 5.6.2 below). (2) The circuit lemma 5.7.1 gives y in D with a circuit of size <= m - 3 for F|x_y=0, or of size <= m - 4 for F|x_y=1 (Lean: `StmtML`, here `stmtML_final`). (3) y is deep (Lean: `deep_mem`): neighbours exactly u1 != u2, each with at most one further neighbour c1, resp. c2; y, u1, u2 are not in A. (4) At y = 0: the restriction is F_{delV G {y}, A} (Lean: `Res.ToolRes0_proof`), with fewer edges and Phi1(G, A) <= Phi1(delV G {y}, A) + 3 (Lean: `Res.ToolDrop0_proof`: the edges at y lie in {yu1, yu2}, and a least cover of the residual plus y covers G, `edges_le`, `tau_delV`). (5) At y = 1: the restriction is F_{delV G {y, u1, u2}, A + {u1, u2}} (Lean: `Res.ToolRes1_proof`), with fewer edges and Phi1(G, A) <= Phi1(residual) + 4 (Lean: `Res.ToolDrop1_proof`: the deleted edges lie in {yu1, yu2, u1c1, u2c2}, a

least cover of the residual plus {u1, u2} covers G, and A gains exactly two elements since u1, u2 are not in A). In either case the induction hypothesis at the residual (A-free by `afree_delV` at y = 0, by Fact 5.6.0 at y = 1) gives Phi1(residual) <= m' + 1 (m' the size of the residual circuit), so Phi1(G, A) <= m + 1. The Lean uses only "at most four edges" in (5); it does not need the four edges to be distinct. QED.

Lemma 5.6.2 (the run gives DeepHyp; `stmtG3`). Let `AFree G A` and let a' a s t b b' be a run. Then `DeepHyp F_{G,A} {a, s, t, b}` holds and {a, s, t, b} has at least two elements. Proof. Write F = F_{G,A}. Only the run is used: no cycle or distance hypothesis enters. H1 (Lean: `nc_of`). For y in D take an edge ij avoiding y and a neighbour c of y (for a: edge st and c = s; for s: edge tb; for t and b: edge as). At the all-one input with x_y := b, F = 1 through ij. At the all-zero input with x_y := b, only y can be on, y is not in A, and no edge has both ends equal to y, so F = 0. H2 (Lean: `nf_of`, applied to each ordered pair of D). For distinct y, y' in D pick a neighbour c of y with c != y' and c not adjacent to y', and a neighbour c' of y' with c' != y (the run's exact neighbour lists and the distinctness of its six vertices give them, for instance c = a' for y = a). With all inputs off except the ones named: F = 1 at (y, y', c) = (1, 0, 1); F = 0 at (0, 0, c) and at (0, 1, c); F = 1 at (y, y', c') = (0, 1, 1); F = 0 at (0, 0, c'). These three inequalities rule out each of the four shapes of a pair funnel. (The working note reads H2 off "no triangle and no 4-cycle"; the Lean reads it off the run alone.) H3 (Lean: `win_of`). For y in D choose c in D adjacent to y all of whose neighbours lie in D (for a and t: c = s; for s and b: c = t). Given x with F(x with D off) = 0, let x' be x with D off and then x_c := 1. Then x' = x off D, x'_y = 0, and F(x') = 0: c is not in A; an edge at c has its other end in D, which is off; any other edge or free input on at x' is on at (x with D off), where F = 0. And F(x' with x_y := 1) = 1 through the edge yc. This is the window pair of the working note. H4 (Lean: `F_mono`). F is monotone and (x with D off) <= x. H5 (Lean: `Forest.F_zero`). F(all zero) = 0. |D| >= 2 since a != s. QED.

5.7 The circuit lemma (`stmtML`)

Theorem 5.7.1 (`StmtML`). Let F be a function on N >= 1 inputs, D a nonempty set of inputs with `DeepHyp F D`, and let F have a circuit with at most s gates. Then for some y in D, F|x_y=0 has a circuit with at most s - 3 gates, or F|x_y=1 has one with at most s - 4 gates. Lean: `H/EdgeFnExt.lean`, `EdgeFn.stmtML`, applied as `stmtML stmtAnd (stmtExt stmtLift stmtXMove stmtFront stmtWin (stmtTwo stmtExch stmtWin))` (`ml_of_front`).

Proof. F is not constant (Lean: `deepHyp_nc`, from H1) and depends on every y in D (Lean: `deepHyp_dep`: H3 at the all-zero input, which is allowed by H5). By the extremal lemma 5.7.2 there is a least clean list C for F of length <= s in which one of three STOP RULES holds: (S1) some y in D has two readers. Then y pays (Lean: `Least.two_readers`, `H/ReadersPat.lean`): if one reader is a NOT gate, at either value it is constant, its own reader J goes too, J is not the other reader (a gate reading both a wire and a gate that reads that wire is a constant or a copy, `Least.no_tri`), so 3 at both values (`Least.third`); if both are binary with the same absorbing value c (0 for AND, 1 for OR), at c both are constant and with their readers 4 go (`Least.fourth`; if the two readers coincide, that gate is constant and its reader goes), and 2 go at the other value (`Least.save2`); if their absorbing values differ, at each value one is constant and 3 go (`Least.third`). Of (3, 3), (4, 2), (2, 4) the lemma keeps 3 at 0, 3 at 0, or 4 at 1. (S2) some gate NOT y, y in D, has two readers. At either value NOT y is constant and its two readers read a constant: 3 at 0 (Lean: `Least.not_two_readers`, through `Least.save3`). (S3) a frontier gate m = AND(l, h), l a literal of y in D, h a gate wire read by m only, m not the output. The output is not h: h is D-free and F depends on y. Then 3 go at y = 0 if l = y, and 4 go at y = 1 if l = NOT y (Lean: `stmtAnd`, Lemma 5.7.6). Each stop rule gives the drop at a size bound <= s (Lean: `Drop.mono`). QED.

Lemma 5.7.2 (extremal choice; `stmtExt`). Under the hypotheses of 5.7.1, some least clean list for F of length <= s satisfies (S1), (S2) or (S3). Proof. A least clean list exists with length <= s (Lean: `Readers.exists_least`, through the transfer of 5.1), and all least lists have the same length L0 (`least_len`). Among least lists choose one, C, whose tail is largest: let M be the largest t <= L0 such that some least list has tail at least t (Lean: `Nat.findGreatest`; every tail is at most the length, `TailGE.le`). Suppose no stop rule holds in C. Then `OneReader D C` holds, so a frontier gate m exists, not the output (Lemma 5.7.3). Write m's operands l (literal of y) and h (D-free). If m is an OR gate: h <= F (Lemma 5.7.4). The lift (Lemma 5.7.5) gives a least list with tail at least M + 1, since m is not tail-OK (Lean: `not_tailOK_or`: if m were an OR below F, then at the value b of y that makes l true, F(x with x_y := b) = 1 for all x, against H1). This contradicts the choice of M. If m is an AND gate: NOT h <= F and h is a gate wire (Lemma 5.7.4). As (S3) fails, h has a reader t != m. By Lemma 5.7.7 h has exactly one other reader r, an AND gate AND(q, h). The exchange move (Lemma 5.7.8), applied with the two AND gates in their order in C, gives a least list with tail at least M + 2, against the choice of M. QED. (The working note proves the circuit lemma by a count over a tree; the Lean follows a remark of that note, the tail potential.)

Lemma 5.7.3 (frontier gate; `stmtFront`, `H/EdgeFnFront.lean`). Let `DeepHyp F D`, D nonempty, C a least clean list for F with `OneReader D C`. Then C has a frontier gate, and it is not the output. Proof. Every input y of D lies in the support of the output (Lean: `front_dep`: H3 at the all-zero input, allowed by H5, gives two inputs differing only at y with different values of F). Suppose some AND or OR gate has support meeting D, and let p be the least such position (Lean: `Nat.find`). Below p (Lean: `front_dlit`): every wire w below wire N + p whose support meets D computes x_y XOR c for some y in D and some constant c, and is the input y or a gate NOT y at a position below p. Proof, in the order the Lean runs it: strong induction on the wire index w (Lean: `Nat.strong_induction_on`).

- w is an input. Its support is {w} (Lean: `front_supp_input`), so w is some i in D; take y = w, c = 0 (Lean: `vl_input`), and w is the input y.
- w is wire N + r with no gate at r. Then its support is empty (Lean: `front_supp_beyond`), against the support meeting D.
- w is wire N + r of a gate g. C is clean (Lean: `Least.hcl`), so g is NOT of one wire a or AND/OR of two wires. If g is AND or OR, its support meets D at a position r below p, against the choice of p as the least such. If g is NOT a: the support of w is the union of the supports of g's operands (Lean: `front_supp_gate`), so the support of a meets D, and a is below w (cleanness); by induction a computes x_y XOR c and is y or a gate NOT y. If a is the input y, then g is the gate NOT y, at position r below p, and w computes x_y XOR (not c) (Lean: `vl_not`). If a is a gate NOT y, then w = NOT NOT y computes the function of the earlier input wire y, which a least list for a non-constant F does not allow (Lean: `Least.no_alias`, with F non-constant); this case does not occur.

The operands j, k of p: at least one has support meeting D (`front_supp_gate`). If both do, both are literals, which is impossible (Lean: `front_two`): for the same input y, j != k (else p copies j), and then y or the gate NOT y has two readers, against `OneReader`; for distinct y, y', every gate reading y, y' or their literals other than through p is excluded by `OneReader`, the output is none of these wires (it would make a restriction at y constant, `front_nlit`), so F sees x_y, x_y' only through p (Lean: `agree_B`), whose value is AND or OR of (x_y XOR c), (x_y' XOR c'); three of the four values of such a gate coincide, so (y, y') is a pair funnel, against H2. So exactly one operand is a literal and the other is D-free: p is a frontier gate. It is not the output (Lean: `front_out`): at the value of y that puts the literal at the gate's absorbing value, F|x_y would be constant, against H1. If no AND or OR gate has

support meeting D: the output, whose support contains some y in D, is then a literal (`front_dlit` up to the end of the list), against H1 (`front_nlit`); or the output wire lies beyond the list and F is constant. QED. (A paper route might rule out "m is the output" with H1 and H5; the Lean uses H1 only.)

Lemma 5.7.4 (window at a frontier gate; `stmtWin` , `H/EdgeFnWin.lean`). In the setting of 5.7.3, let m be a frontier gate with operands l (literal of y) and h. If m is OR, then h = 1 forces F = 1. If m is AND, then h = 0 forces F = 1, and h is a gate wire. Proof (Lean: `win_core`). Let c be the absorbing value of m (1 for OR, 0 for AND) and suppose h(x) = c and F(x) = 0. By H4, F(x with D off) = 0; by H3 there is x' equal to x off D with F(x') = 0, F(x' with x_y := 1) = 1. The wire h is D-free, so it has the value c at x' and at x' with x_y := 1 (`vl_dfree`), and m has the value c at both (`vl_abs`). Every gate other than m that reads y or l is excluded by `OneReader` (if l = NOT y, the only reader of y is that NOT gate, and its only reader is m). The output is neither y nor l (`out_ne_lit` , from H1). So every wire other than y and l, the output included, agrees at the two points (Lean: `agree_B`), and F(x') = F(x' with x_y := 1), a contradiction. For AND: if h were an input wire, it would be 0 at the all-zero input, forcing F(all zero) = 1, against H5. Only cleanness and correctness of C are used, not its leastness. QED.

Lemma 5.7.5 (the lift; `stmtLift` , `H/EdgeFnLift.lean`). Let C be a least clean list for a non-constant F, and let gate p of C be OR(l, h) with h <= F. Delete p, let its readers read l, and append OR(output, h). The result is a least list for F; if p was not tail-OK, its tail grows by one. Proof. Where h = 0, gate p equals l, so after the redirect every wire keeps its value up to renumbering (Lean: `rdel_all` , piece C1 `stmtRD` , `H/EdgeFnModel.lean`), and the appended gate computes the old output, F. Where h = 1, F = 1 and the appended OR is 1. The list has the same length (`rdel_length`) and is clean (`rdel_clean`), so it is least (`least_of`). For the tail (Lean: `tailGE_rdel_app` with the set {h = 0}): p is not tail-OK, so it stood before the tail (`TailGE.lt_of_not`); each old tail gate keeps its operation, and its value where h = 0, while where h = 1, F = 1, so "below F" survives; the appended OR computes F, so it is tail-OK. QED.

Lemma 5.7.6 (an unshared AND partner pays; `stmtAnd` , `H/EdgeFnAnd.lean`). Let C be a least clean list for F with both restrictions at y not constant and F depending on y; let gate p = AND(l, h) with l a literal of y and h the wire of a gate at position ph (so h is the value and ph the place), read only by p; let the output be neither p nor h. Then 3 gates go at y = 0 if l = y, and 4 go at y = 1 if l = NOT y. Proof. The Lean does not delete h; it replaces gate ph by NOT y (Lean: `andSwap`), which keeps the list clean and of the same length (`andSwap_clean` , `andSwap_length`). Let b be the value of y at which l is 0 (b = 0 for l = y, b = 1 for l = NOT y). Wherever l = 0, gate p is 0 before and after the change, and p is the only reader of ph, so every wire other than ph keeps its value (Lean: `andSwap_agree`); in particular the changed list computes F on {x_y = b}. p is not the output, so it has a reader S (Lean: `Least.live`); l != h (else p copies h, `Least.no_alias`). In the changed list, at y := b, the positions ph (reads y), p (reads the constant l), S (reads p, now constant 0) read a wire of constant status; for l = NOT y the NOT gate pn on y does too. The count `Readers.save` (5.1, through Lemma 5.7.10; Lean: `and_core`) removes these 3, resp. 4, positions, and the restriction is not constant by hypothesis. QED. (A person would say: at y = b, p is constant, its reader goes, and h, read only by p, dies; the Lean obtains the same count by turning h into a gate on y.)

Lemma 5.7.7 (the second reader of an AND partner; `stmtTwo` , from `stmtExch` and `stmtWin` , `H/EdgeFnTwo.lean`). In the setting of 5.7.3, let the frontier gate p = AND(l, h) and let h have a reader other than p. Then h has exactly one other reader r, and r is an AND gate AND(q, h). Proof. By 5.7.4, NOT h <= F. The strict exchange count (Lemma 5.7.9) applies to every set T of positions each of which reads h, or reads an OR or NOT gate that reads h: some clean list for F has length <= |C| + 2 -

|T|. As C is least, |T| <= 2 for every such T. Three readers of h would form such a T: so every reader of h is p or r. r is not a NOT gate: if r = NOT h were the output, then F = NOT h, but h does not depend on y (`vl_update_of_not_supp`) while F does (H3 at the all-zero input, H5); if not the output, r has a reader t (`Least.live`), t != p (p reads only l and h, and r's wire is neither), and {p, r, t} is such a T. r is not an OR gate: if it were the output, F would be constant 1 (where h = 0 by NOT h <= F, where h = 1 since r reads h); if not, its reader t gives {p, r, t} as before. So r is an AND gate reading h and some wire q. QED.

Lemma 5.7.8 (the exchange move; `stmtXMove`, `H/EdgeFnXMove.lean`). Let C be a least list for a non-constant F, gates p < p' be AND(l, h) and AND(q, h), and NOT h <= F. Delete p' (its readers read q), then p (its readers read l), and append NOT h and OR(output, NOT h). The result is a least list for F whose tail is at least the old tail plus 2. Proof. Where h = 1 both deleted gates equal their redirect targets (`vl_and_true`), so the old output survives the two moves (piece C1); where h = 0, the appended OR is 1 and so is F. The length is unchanged, so the list is least (`least_of`). An AND gate is never tail-OK (`not_tailOK_and`), so both deleted gates stood before the tail, and `tailGE_rdel_app` applied twice, with the appended NOT and the appended OR (which computes F) both tail-OK, gives tail + 2. QED. (This is the NOT exchange of the proof: the two AND gates on h are traded for one NOT h and one OR. The working note uses a lemma of its own here; the Lean uses this explicit move, and that lemma only in the count 5.7.9.)

Lemma 5.7.9 (the strict exchange count; `stmtExch`, `H/EdgeFnExch.lean`). Let C be a clean list computing a non-constant F, h a wire with NOT h <= F, and T a set of positions each of which reads h, or reads an OR or NOT gate that reads h. Then some clean list for F has length <= |C| + 2 - |T|. Proof. h is 1 somewhere (else F would be constant 1) and is a wire of C. Keep the gates up to and including h; run the simplification pass of `H/Elim.lean` on the rest with the status of h set to the constant 1 (as in `H/ExchangeElim.lean`); its invariant holds at every input where h = 1 (`inv_elim`, `Exchange.inv_start`). Each position of T reads h, of constant status, or reads an OR or NOT gate reading h, whose status is then constant too (`exch_const_reader`); the count `Helpers.elim_count` (Lemma 5.7.10, started from the kept gates with h of constant status, Lean: `Exchange.startStats`) removes |T| gates. The status of the output decides the list: constant 1 is impossible (F would be constant); constant 0 gives F = NOT h (the kept gates and NOT h); a wire j gives the simplified gates, NOT h and OR(j, NOT h). In every case the length is at most |C| - |T| + 2, and the list computes F: where h = 1 by the invariant, where h = 0 because F = 1 and the top OR (or NOT h) is 1. QED.

Lemma 5.7.10 (the count; `Helpers.elim_count`, `H/Parity3.lean`). Run the cleaning pass of section 3.1 over a list gs of gates, each a NOT of one wire or an AND/OR of at most two (Lean: `NOk`), starting from a state with a list of statuses and a list of already emitted gates. Let S be a set of positions of gs, each holding a gate that reads an earlier wire whose final status is a constant. Then (gates emitted at the end) + |S| <= (gates emitted at the start) + |gs|. Proof, in the order the Lean runs it: induction on gs from its last gate (Lean: `List.reverseRecOn`).

- gs empty. S has no position to hold, so S is empty, and both sides are equal.
- gs = gs0 followed by one gate g. The pass over gs is the pass over gs0 followed by one step on g (Lean: `elim_append`), and that last step does not change the status of any wire below g (Lean: `elim_split_getD_lt`). So S with the position of g removed satisfies the hypothesis for gs0, and the induction hypothesis gives the count for it. If the position of g is in S: g reads a wire of constant status, so its step emits no gate (Lean: `elimStep_none`, the cleaning pass of section 3.1), and removing that position lowered |S| by exactly one (Lean: `Finset.card_erase_of_mem`). If it

is not in S: the step emits at most one gate (Lean: `toList_length_le`) and S is unchanged. In both cases the inequality for gs follows by adding one to each side (Lean: `omega`).

`Readers.save` (5.1) applies it from the start state with input i of constant status b and no emitted gate (Lean: `statAt`), so the pass emits at most |gs| - |S| gates; 5.7.9 applies it from the kept gates with h of constant status 1.

5.8 Assembly of 5.0.1

`stmtFront` proves the one hypothesis of `ml_of_front`, which yields `StmtML` (5.7.1); `lb_of_ml` feeds it, with `stmtG3` (5.6.2) and `stmtG4` (5.4.2), to `lb_of_pieces`, which applies `a1_of_pieces` (5.3.1) with `stmtG1` (5.4.1), `stmtG6` (5.5.1) and the residual and drop tools of `H/LocRes.lean`, and then `lb_of_a1` (5.2). The row statement is `StmtLB` verbatim (Lean: `EdgeFn.stmtLB_final`). This proves Theorem 5.0.1.

5.9 NOT gates do not help (Theorem 5.0.3)

Proof (`L113.lean`, in the order of the file). Let C be a fan-in-two circuit computing f_G. G has an edge. Take a least vertex cover T (Lean: `Finset.exists_min_image` over the covers). Since degree and neighbour-set size agree, Theorem 5.0.1 applies with A empty and k = |T|, the hypotheses 9 weakened to 7 (Lean: `PEqualNP.MetaComplexity.L204` at A = empty set, k = T.card, s = size(C)): |E| + |T| <= size(C) + 1. The cover circuit of section 2 without NOT gates gives D with no NOT gate, computing f_G, with size(D) + 1 <= |E| + |T| (Lean: `NotFreeCor.nfc_edge_upper_mono`, a copy of the construction of L-meta-154, `H/NotFreeCor.lean`, copied only to avoid a name clash between imports). So size(D) <= size(C), and D computes the function of C. G has no edge. D is the empty circuit with output wire n, which does not exist and reads 0; f_G is constant 0. QED. The hypotheses of 5.0.3 enter the proof only through Theorem 5.0.1, after weakening 9 to 7.

5.10 The monotone bound (Theorem 5.0.2): cited

Theorem 5.0.2 (L-meta-115) has its own Lean proof (`Loc.Final.row`, `H/LocFinal.lean`, which runs `Assemble.row_of_last`, `H/LocAssemble.lean`, and then `loc_of_pieces`, `H/LocPlan.lean`; circuits without NOT gates on the class with cycles of length at least 9 and hub walks at least 7). It shares with the proof of 5.0.1 the clean-list model and its basic evaluation and counting lemmas (among them `vl_gate`, `vl_input`, `agree_B`, `reads_lt`, `PermInputs.dep_lower_set`, `inv_elim`, `elim_len`, `relabel_size`), but none of the circuit lemmas of 5.5 to 5.7, and it imports no EdgeFn file; the proof of 5.0.1 also shares graph and restriction lemmas with it (`H/LocRes.lean`, `H/LocStruct.lean`, `H/LocFree.lean`). This document cites that proof and does not reproduce it. On paper, not a row: its class lies inside the class of 5.0.1 (cycles at least 9 >= 7, hub walks at least 7), and Theorem 5.0.1 with A empty and k = tau(G), which holds for every circuit and so for circuits without NOT gates, gives the statement of 5.0.2 in full with T a least cover (T empty when G has no edge), the same combination the proof of 5.0.3 makes in `L113.lean`. The paper text its own proof follows is an internal working note.

6. Where the method stops: the 4-cycle

Let C4 be the 4-cycle on vertices 0, 1, 2, 3 in this order, so

f(x) = x_0 x_1 OR x_1 x_2 OR x_2 x_3 OR x_3 x_0.

It has |E| = 4 and tau(C4) = 2, so |E| + tau(C4) - 1 = 5. But

f(x) = (x_0 OR x_2) AND (x_1 OR x_3),

since expanding the right side gives x_0 x_1 OR x_0 x_3 OR x_2 x_1 OR x_2 x_3, the four edges. This is a circuit of 3 gates: OR(x_0, x_2), OR(x_1, x_3), AND of the two. So the inequality |E| + tau(C4) - 1 <= size fails on C4 by two gates. (The expansion is on paper. The row L-meta-112 states that f has a circuit with s gates iff 3 <= s.) Lean: `lean/PEqualNP/MetaComplexity/L112.lean`, `PEqualNP.MetaComplexity.L112` (proof term `PEqualNP.MetaComplexity.Helpers.ExactCycle4.size_iff`, `H/ExactCycle4.lean`); checked as L-meta-112, run-011875.

C4 is not a forest, so Theorem 4.1 does not apply to it. What follows says which step of the argument of section 4 breaks if one tries to run it on C4, and why it cannot be patched by a different choice of vertex. All of it is on paper.

The step that fails: the existence of a good leaf (Lemma 4.2), and with it Tool C. C4 has no leaf: every vertex has two neighbours. The induction of 4.4 has no other step for a graph with an edge, so it cannot start. The two restriction identities of Tool C both use that l is a leaf (`restrict_false` and `restrict_true` take "every neighbour of l is p" as a hypothesis). At a vertex v of C4 that is not a leaf, say v = 0 with neighbours 1 and 3 (Phi is the potential of the end of section 4.4, at k = tau):

- x_0 := 0 deletes two edges, 01 and 03, not one: the residual is the path 1 - 2 - 3, with Phi = 2 + 1 - 1 = 2. Phi drops from 5 to 2, by 3.
- x_0 := 1 turns both neighbours 1 and 3 into free inputs and deletes every edge at 1 or 3: the residual is x_1 OR x_3 (the edges 12 and 23 are absorbed), with Phi = 2 + 0 + 0 - 1 = 1. Phi drops from 5 to 1, by 4.

(The formula of `restrict_true`, used without its leaf hypothesis at l = 0, p = 1, q = 2, would give F_{G - {01, 12}, {1}}, which still contains the edge 03; it is not the restriction.)

Why no choice of vertex rescues it. The forest argument works because at a good leaf the potential drops by at most 2 (or by 1) and the cleaning pass removes at least as many gates. On C4 every vertex looks alike, and the drops are 3 at false and 4 at true. In the 3-gate circuit each input is read once, by an OR gate:

- x_0 := 0: the reader OR(x_0, x_2) becomes an alias of x_2; one gate goes (3 to 2), against a drop of 3;
- x_0 := 1: OR(x_0, x_2) becomes the constant 1 and the AND becomes an alias of OR(x_1, x_3); two gates go (3 to 1), against a drop of 4.

So the inequality "gates removed >= drop of Phi", which every step of 4.4 establishes, is false at every vertex of C4 at both values: the circuit saves 1 at false and 2 at true where the bound would need 3 at false or 4 at true. The cause is the sharing in (x_0 OR x_2) AND (x_1 OR x_3): two opposite vertices of the 4-cycle have the same two neighbours, so one OR gate on x_0, x_2 serves all four edges at once. In the terms of section 5 (on paper): the opposite pair x_0, x_2 is a pair funnel of f, since the three restrictions (x_0, x_2) = (1, 0), (0, 1), (1, 1) all equal x_1 OR x_3, so H2 of 5.1 fails on C4. The Lean of L-meta-204 does not exclude C4 at H2: it excludes it through the class T7 of 5.4 (every cycle of length at least 7), and C4 has no good leaf and no run (a run needs six distinct vertices). The cycle C4 is the cycle function C_n at n = 4, where the formula n + ceil(n/2) - 1 of Theorem 3.6 would give 5 while 3 gates suffice: the formula is false at n = 4. Theorem 3.6 assumes n >= 5.

7. Prior work

The function. For a graph G, the edge function f_G(x) is the OR over edges uv of (x_u AND x_v). It is the quadratic Boolean function of G, studied under the name of graph complexity. The model here counts every gate of a fan-in-two circuit over AND, OR and NOT, including NOT gates.

Monotone formulas and single-level circuits. For a graph with no triangle and no 4-cycle, every monotone formula for f_G has at least |E| leaves (Jukna, "Computational Complexity of Graphs", author's draft of a chapter for an edited volume, 2013, Theorem 1.23; the same statement is Theorem 7.2 of the author's open draft of "Boolean Function Complexity", 2012). For graphs with no 4-cycle, single-level monotone circuits need |E| gates (Amano and Maruoka, 2006, as quoted in Jukna, ECCC TR05-021; the 2006 paper itself not opened). Jukna, ECCC TR05-021 ("Disproving the single level conjecture", read in full), shows that single-level bounds do not transfer to general monotone circuits, and poses as open a superlinear bound for monotone circuits computing f_G for an explicit G (Problem 24).

Restricted monotone models. For a bipartite graph with no 4-cycle, f_G needs |E|/2 gates in multilinear monotone Boolean circuits, in tropical circuits and in monotone arithmetic circuits; over general monotone Boolean circuits the same source notes that no such bound was known (Jukna, "Lower bounds for tropical circuits and dynamic programs", arXiv:1406.3065, 2014, Remark 10 with Theorems 9 and 18, read at source).

Exact values. For the complete graph (threshold-2), exact counts are printed: monotone circuit size above 2n - 4 + log n, with AND count ceil(log n) and OR count 2n - 4 (Wegener, "The Complexity of Boolean Functions", 1987, Theorem 6.3.1, read; after Bloniarz 1979, not opened). For parity in the basis AND, OR, NOT, the least number of AND and OR gates is 3(n - 1) (Schnorr 1974, as proved in Jukna's 2012 book draft, Theorem 1.29, read); there NOT gates are not counted.

Gate elimination. The method of this note is gate elimination. Its general form gives about 2n for functions whose pairs of inputs leave three distinct subfunctions (Jukna's 2012 book draft, Theorem 1.28, read); the strongest explicit bounds by it are (3 + 1/86)n over all two-input gates (Find, Golovnev, Hirsch and Kulikov, FOCS 2016, read) and about 3.1n (Li and Yang, 2022, abstract only), and 5n - o(n) in the De Morgan basis (Iwama, Lachish, Morizumi and Raz, 2002, as cited in two 2026 arXiv papers, read). Its limits: on functions composed with stable gadgets, an induction of this kind certifies only O(n) (Golovnev, Hirsch, Knop and Kulikov, "On the limits of gate elimination", MFCS 2016, read). The bounds of this note are linear: the least size is at most the value |E| + |T| - 1 for any vertex cover T (L-meta-105, L-meta-154), which on the sparse classes treated is linear in the number of inputs (on paper).

Boolean sums and NOT gates. The several-output relative, the system of ORs of sets meeting pairwise in at most one element, needs exactly sum (|S_i| - 1) monotone gates (Wegener 1987, Lemma 6.1, Theorem 6.2, Corollary 6.1, read; after Mehlhorn 1979 and Nechiporuk, not opened); with NOT gates, some dense Boolean sums are much cheaper (Boolean convolution, Wegener 1987, Section 6.7, read). For the single-output f_G, the row L-meta-113 says NOT gates do not lower the least size on cycles at least 9 with hubs at least 9 apart; on forests, paths, cycles with n >= 5 and the class of L-meta-204 the same is on paper, not checked, read off L-meta-104, L-meta-204, L-meta-102 and L-meta-106 with the circuit of L-meta-154 (section 0).

What we do not know

We do not know whether these values are already in print. Two independent searches (about twenty queries; sources above) found no printed statement of the exact value |E| + tau(G) - 1 for f_G with NOT gates allowed, on forests, cycles or graphs with long threads, nor one that implies it in a few lines. NOT read, any of which may contain it or part of it: Knuth, The Art of Computer Programming, Volume 4A, Section 7.1.2 (exhaustive optimal circuits for at most five inputs); the published text of Jukna's 2012 book; Amano and Maruoka, Algorithmica 46 (2006) (no open copy found); Lenz and Wegener (1991), Mirwald and Schnorr (1992), Pudlak, Rodl and Savicky (1988), each known only through quotation; Bloniarz (1979); Krichevskii (1964); Hansel (1964); the 1970s literature on exact minimality in the AND, OR, NOT basis.

8. Table: every step, its Lean file and declaration

Paths are under `lean/PEqualNP/MetaComplexity/` (H = `Helpers/`). Line numbers are omitted on purpose: names are stable, lines are not.

step	declaration	file
1.1, 1.2 root circuits, size	`GateOp`, `Gate`, `Circuit`, `Circuit.wires`, `Circuit.eval`, `Circuit.size`	`lean/PEqualNP/Defs/CircuitLowerBounds.lean`
1.3 fan-in two, circuit size	`FanInTwo`, `HasCircuitSize`, `circuitFun`	`lean/PEqualNP/Defs/MetaComplexity.lean`, `MetaComplexityMinimal.lean`
1.5 to 1.7 F, covers, tau, Phi1	`Forest.F`, `Forest.Cov`, `Loc.AFree`, `Loc.tau`, `Loc.tau_attained`, `Loc.tau_le`, `Loc.Phi1`	H/ForestBound, H/LocGraph
2 upper bound	`EdgeUpper.edge_upper`, `EdgeUpperMono.edge_upper_mono`	H/EdgeUpper, H/EdgeUpperMono
2 building steps	`Bld`, `bld_var`, `bld_bin`, `bld_and`, `bld_or`, `bld_mono`, `bld_congr`, `bld_anyList`, `bld_anyFinset`, `bld_edges` (with its local fact `key`), `bld_hasCircuitSize`, `bld_circuit`; gate condition `Ok`	H/EdgeUpper, H/EdgeUpperMono; `Ok` in H/Shannon
3.1 Lemma 3.1 counting	`PermInputs.dep_lower_set`; `dep_lower`, `inputSupps`, `gateSupp`, `suppsFrom`, `agree_wiresFrom`, `Cnt`, `cnt_inputs`, `cnt_step`, `cnt_suppsFrom`	H/PermInputs, H/Support
3.1 Lemma 3.2 renaming	`PermInputs.relabel_size`, `wmap`, `relabel`, `rel_fold`, `rel_init`, `snoc_swap`	H/PermInputs
3.1 cleaning pass	`gate_elim`, `normGate`, `Stat`, `NOk`, `elim`, `elimStep`, `simplify`, `simpAnd`, `simpOr`, `simpNot`, `simplify_count`, `inv_elim`, `elim_len`, `elim_len_lt`, `supp_noread`	H/Elim

step	declaration	file
3.1 cleaning pass, constants	`elimStep_none`, `elimStep_const`, `simplify_const`, `fixBit`, `supp_noreadD`, `elim_append`	H/Parity
3.1 Lemma 3.3(a) one gate	`Forest.one_gate_at`	H/ForestElim
3.1 Lemma 3.3(b) two gates	`PermInputs.gate_elim_at`; `gate_elim_two`, `two_saved`	H/PermInputs, H/Parity
3.2 paths	`path_size`, `path_upper`, `path_lower`, `G_lower`, `G_dep`, `G_del_false`, `G_del_true`, `del`, `path_eq_G`	H/PathSizeUpper, H/PathSizeLower
3.2 path circuit	`pathGates`, `pathB`, `pathB_step`, `pathB_ofFn`, `three_gates`	H/PathSizeUpper
3.3 Theorem 3.5 (L-meta-107)	`L107` = `Readers.fanout_step_3_4`	L107.lean, H/ReadersExport
3.3 cycles, core	`cyc_exact_of_readers`, `cyc_core_of_readers`, `StmtCycCore`, `cyc_dep`, `cyc_nonconst`, `cyc_no_funnel`, `exists_nbr`, `cyc_sens`, `size_mono`	H/CycleCore, H/CycleUpper
3.3 cycles, lower and upper	`cyc_exact_of_core`, `restrict_last`, `rot`, `cyc_rot`, `piv`, `cyc_snoc_false`, `cyc_snoc_true`, `cyc_false_lower`, `cyc_true_lower`, `arith_false`, `arith_true`, `cyc_upper`, `cyc_eq_path_or`; path gates `CycleSize.PG.pathGates`	H/CycleUpper, H/CyclePathGates
4.1 forests	`Forest.claimQ`, `Forest.forest_bound`	H/ForestBound
4.2 Tool C	`Forest.restrict_false`, `Forest.restrict_true`, `ncard_del1`, `ncard_del2`, `cov_insert`, `cov_swap`, `Forest.F_dep`, `F_dep_A`, `Forest.F_nc`, `Forest.F_zero`, `del_p_adj`, `Forest.noedge_ncard`	H/ForestBound
4.3 forest good leaf	`Forest.forest_good_leaf`, `exists_longest_path`	H/ForestGraph
5.1 clean lists, least lists	`CleanL`, `vl`, `Readers.Least`, `Readers.Reads`, `Readers.PairFunnel`	H/Parity3, H/Parity4, H/Readers
5.1 transfer	`clean_of`, `Readers.exists_least`, `Readers.has_of_clean`	H/Parity3, H/Readers

step	declaration	file
5.1 reduced (f non-constant)	`Least.no_const`, `no_alias`, `live`; `no_tri`	H/Readers; H/ReadersPat
5.1 delV, BoundOf	`Loc.delV`, `Loc.BoundOf`	H/LocGraph
5.1 run, deep vertex	`Loc.Run`, `Loc.Deep`, `Run.deep_a`, `deep_s`, `deep_t`, `deep_b`, `deep_mem`	H/LocGraph, H/EdgeFnGraphPlan
5.1 literals, frontier, tail, rdel, deep hypotheses	`DeepHyp`, `IsLit`, `DFree`, `OneReader`, `Front`, `TailOK`, `TailGE`, `rdel`, `rd`, `StmtLB`	H/EdgeFnPlan
5.1 drops, the count	`Drop`, `Pays`, `Readers.save`, `RC`, `statAt`, `passG`	H/Readers, H/Parity3, H/Parity4
5.2 class to row	`EdgeFn.lb_of_a1`	H/EdgeFnGraphPlan
5.3 induction	`a1_of_pieces`, `Hered`, `Dich`, `afree_delV`	H/EdgeFnGraphPlan
5.4 class, good leaf	`T7`, `GoodLeaf`	H/EdgeFnGraphPlan
5.4.1 heredity	`stmtG1`, `delV_le`	H/EdgeFnGraphPlan, H/LocGraph
5.4.2 leaf or run	`stmtG4`, `thread7`, `dich_of_g4`; `Loc.Struct.Good`, `exists_max`, `two_other`, `classify`, `two_three`, `run_of_thread`	H/EdgeFnGraphPlan, H/LocStruct
5.5 leaf step	`stmtG6`, `leaf_drop0`, `leaf_drop1`	H/EdgeFnGraphPlan
5.5, 5.6 restrictions, drops	`Loc.ToolRes0`, `ToolRes1`, `ToolDrop0`, `ToolDrop1`; proofs `Res.ToolRes0_proof`, `Res.ToolRes1_proof`, `Res.ToolDrop0_proof`, `Res.ToolDrop1_proof`, `edges_le`, `tau_delV`	H/LocPlan, H/LocRes
5.6 deep step	`deep_step`, `deep_mem`	H/EdgeFnGraphPlan
5.6.2 deep hypotheses	`stmtG3`, `nc_of`, `nf_of`, `win_of`, `F_mono`	H/EdgeFnGraphPlan
5.7.1 circuit theorem	`StmtML`; `EdgeFn.stmtML`, `deepHyp_nc`, `deepHyp_dep`; `stmtML_final = ml_of_front stmtFront`	H/EdgeFnPlan, H/EdgeFnExt, H/EdgeFnFinal, H/EdgeFnOfFront
5.7.1 (S1), (S2) pay	`Readers.Least.two_readers`, `not_two_readers`, `Least.third`, `Least.fourth`; `Least.save2`, `Least.save3`; `Drop.mono`	H/ReadersPat; H/Readers

step	declaration	file
5.7.2 extremal choice	`StmtExt` in H/EdgeFnPlan; `stmtExt`, `least_len`, `not_tailOK_or`; `TailGE.le`, `TailGE.lt_of_not`	H/EdgeFnExt, H/EdgeFnModel
5.7.3 frontier gate	`StmtFront`	H/EdgeFnPlan
5.7.3 frontier gate, proof	`stmtFront`, `front_dep`, `front_dlit`, `front_supp_input`, `front_supp_beyond`, `front_supp_gate`, `front_two`, `front_out`, `front_nlit`; `vl_input`, `vl_not`, `agree_B`	H/EdgeFnFront; H/Parity4, H/Readers
5.7.4 window	`StmtWin`	H/EdgeFnPlan
5.7.4 window, proof	`stmtWin`, `win_core`, `vl_dfree`, `out_ne_lit`; `vl_abs`	H/EdgeFnWin; H/Parity4
5.7.5 lift	`StmtLift`, `rdel`	H/EdgeFnPlan
5.7.5 lift, proof	`stmtLift`; `stmtRD`, `rdel_all`, `rdel_length`, `rdel_clean`, `least_of`, `tailGE_rdel_app`	H/EdgeFnLift, H/EdgeFnModel
5.7.6 unshared partner	`StmtAnd`	H/EdgeFnPlan
5.7.6 unshared partner, proof	`stmtAnd`, `andSwap`, `andSwap_clean`, `andSwap_length`, `andSwap_agree`, `and_core`	H/EdgeFnAnd
5.7.7 to 5.7.9 shared partner	`StmtTwo`, `StmtXMove`, `StmtExch` in H/EdgeFnPlan; `stmtTwo`, `vl_update_of_not_supp`; `stmtXMove`, `not_tailOK_and`, `vl_and_true`; `stmtExch`, `exch_const_reader`	H/EdgeFnTwo, H/EdgeFnXMove, H/EdgeFnExch
5.7.9 start of the pass	`Exchange.startStats`, `Exchange.inv_start`, `Exchange.elim_prefix`	H/ExchangeElim
5.7.10 the count	`Helpers.elim_count`, `elim_split_getD_lt`, `elimStep_none`, `toList_length_le`	H/Parity3, H/Parity, H/Elim
5.8 assembly	`lb_of_pieces`, `lb_of_ml`, `lb_of_front`, `stmtLB_final`, `L204`	H/EdgeFnGraphPlan, H/EdgeFnOfFront, H/EdgeFnFinal, L204.lean
5.9 NOT gates	`L113` (in file), `NotFreeCor.nfc_edge_upper_mono`	L113.lean, H/NotFreeCor
5.10 monotone	`Loc.Final.row`, `Assemble.row_of_last`, `loc_of_pieces`, `main_of_pieces`	H/LocFinal, H/LocAssemble, H/LocPlan

step	declaration	file
6 four-cycle	`ExactCycle4.size_iff`, `up`, `lower`	H/ExactCycle4

Notes on the files, spot-checked against them: no `sorry`, `axiom` or `admit` appears in this chain. Files in the folder that L-meta-204 does NOT import: `EdgeFnFrontB` (a second proof of `StmtFront`), the `EdgeFnBase*` files, `EdgeFnCyc`, `EdgeFnRun4`, `EdgeFnRun4Row`. Two docstrings are older than the proofs they describe (`CycleUpper.lean` calls `StmtCycCore` open, see the end of section 3.3; `EdgeFnGraphPlan.lean` calls `StmtML` its open hypothesis); both are proved (`cyc_core_of_readers`, `stmtML_final`). `EdgeFnExch.lean` imports, through `H/ExchangeElim.lean`, a draft file of unproved `Prop` definitions (not axioms); from that chain it uses only `Exchange.startStats`, `Exchange.inv_start`, `Exchange.elim_prefix` and `Exchange.startStats_getD` of `H/ExchangeElim.lean`, and nothing from the draft file itself; its count is `Helpers.elim_count` of `H/Parity3.lean`.