Simple Circuit Extensions for XOR in PTIME

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

The Minimum Circuit Size Problem for Partial Functions (MCSP^*) is hard assuming the Exponential Time Hypothesis (ETH) (Ilango, 2020). This breakthrough hardness result leveraged a characterization of the optimal $\{\wedge, \vee, \neg\}$ circuits for n-bit OR (OR_n) and a reduction from the partial f-Simple Extension Problem where $f = \mathsf{OR}_n$. It remains open to extend that reduction to show ETH-hardness of total MCSP. However, Ilango observed that the total f-Simple Extension Problem is easy whenever f is computed by read-once formulas (like OR_n). Therefore, extending Ilango's proof to total MCSP would require one to replace OR_n with a slightly more complex but similarly well-understood Boolean function.

This work shows that the f-Simple Extension problem remains easy when f is the next natural candidate: XOR_n. We first develop a fixed-parameter tractable algorithm for the f-Simple Extension Problem that is efficient whenever the optimal circuits for f are (1) linear in size, (2) polynomially "few" and efficiently enumerable in the truth-table size (up to isomorphism and permutation of inputs), and (3) all have constant bounded fan-out. XOR_n satisfies all three of these conditions. When $\neg$ gates count towards circuit size, optimal XOR_n circuits are binary trees of $n-1$ subcircuits computing $(\neg)\mathsf{XOR}_2$ (Kombarov, 2011). We extend this characterization when $\neg$ gates do not contribute the circuit size. Thus, the XOR-Simple Extension Problem is in polynomial time under both measures of circuit complexity.

We conclude by discussing conjectures about the complexity of the f-Simple Extension problem for each explicit function f with general circuit lower bounds over the DeMorgan basis. Examining the conditions under which our Simple Extension Solver is efficient, we argue that *multiplexer* functions (MUX) are the most promising candidate for ETH-hardness of a Simple Extension Problem, towards proving ETH-hardness of total MCSP.

2012 ACM Subject Classification Theory of computation → Circuit complexity; Theory of computation → Fixed parameter tractability

Keywords and phrases Minimum Circuit Size Problem, Circuit Lower Bounds, Exponential Time Hypothesis

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

© Anonymous author(s);
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Circuits model the computation of Boolean functions on fixed input lengths by acyclic wires between atomic processing units — logical "gates." To measure the circuit complexity of a function f, we first fix a set of gates $\mathcal{B}$ — called a *basis*. This work studies circuits over the following basis: fan-in 2 AND, fan-in 2 OR, and fan-in 1 NOT gates. We consider two complexity size measures $\mu_{\mathcal{D}}$ and $\mu_{\mathcal{R}}$, which count *only* the binary gates and the *total* number of gates in a circuit respectively. We will refer to $\mathcal{B}$ equipped with these two complexity measures as $\mathcal{D}$, the DeMorgan basis, and $\mathcal{R}$, the Red'kin basis respectively[1].

Basic questions about these models have been open for decades; we cannot even rule out the possibility that every problem in NP is decided by a sequence of linear-size circuits (see page 564 of [22]). Despite this, the ongoing search for circuit complexity lower bounds has fostered rich and surprising connections between cryptography, learning theory, and algorithm design [16, 12, 37, 6, 36, 17]. The *Minimum Circuit Size Problem* (MCSP, [23]) appears in all of these areas, asking:

> Given an n-input Boolean function f as a 2^n-bit truth table, what is the minimum s such that a circuit of size s computes f ?

The **existential** question — do functions that require "many" gates exist? — was solved in 1949: Shannon proved that almost all Boolean functions require circuits of near-trivial[2] size $\Omega(\frac{2^n}{n})$ by a simple counting argument [40]. The current best answer to the **explicit** question in the DeMorgan basis — is such a hard function in NP? — is a circuit lower bound of $5n - o(n)$, proved via *gate elimination* [21]. This is far from the popular conjecture that NP-complete problems require super-polynomial circuit size.

The **algorithmic** question — is MCSP NP-hard? — remains open after nearly fifty years [41], even under strong complexity assumptions such as the Exponential Time Hypothesis (ETH). But, many natural variants of MCSP have been proven NP-hard unconditionally. For instance, DNF-MCSP [31], MCSP for OR-AND-MOD Circuits [15], and MCSP for multi-output functions [20] are now known to be NP-hard. Furthermore, MCSP for partial functions (MCSP*) [18] is hard under the Exponential Time Hypothesis (ETH), later extended to unconditional NP-hardness under randomized reductions [14].

Our work studies the feasibility of generalizing Ilango's technique for ETH-hardness of MCSP* to total MCSP. In particular, underlying Ilango's proof is a related decision problem about circuit complexity of Boolean *simple extensions* which we call the f-Simple Extension Problem (f-SEP).

▶ **Definition** (Simple Extension). *Let f be a Boolean function that depends on all of its n variables. A simple extension of f is either f itself or a function g on $n + m$ variables satisfying:*

1. *g depends on all of its inputs.*
2. *$CC(g)$ — the circuit-size complexity of g — is $CC(f) + m$.*
3. *There exists a setting $k \in \{0,1\}^m$, a* key*, such that for all $x \in \{0,1\}^n$, $g(x,k) = f(x)$.*

We define the f-Simple Extension decision problem for *total functions* below.[3]

---

[1] We will specify a basis if a statement pertains to *only* that basis. If the basis is not specified, then the statement applies to both $\mathcal{D}$ and $\mathcal{R}$.

[2] From using a lookup table.

[3] For partial function f-Simple Extension (f-SEP*), g is a *partial* function and we must determine whether any completion of g is a simple extension of f.

▶ **Problem** (The f-Simple Extension Problem). *Let f be a sequence of Boolean functions $\{f_n\}_{n\in\mathbb{N}}$ such that each f_n depends on all of its n inputs. The f-Simple Extension Problem is defined as follows: Given $n \in \mathbb{N}$ and $tt(g)$—the truth table of a binary function g—decide whether g is a simple extension of f_n.*

For a fixed f whose truth table can be efficiently computed and whose exact circuit complexity is known, f-SEP reduces to a single call to an MCSP oracle, because checking whether g is a non-degenerate extension of f can be done in polynomial time via brute force (given the truth table of g). This observation gives rise to an MCSP-hardness proof framework: if one can identify an explicit function f for which deciding the f-Simple Extension Problem is hard, then MCSP is also hard.

This framework was implicitly used in Ilango's hardness proof for MCSP* [18], i.e. reducing an ETH-hard problem to deciding whether a partial function is a simple extension of $f = \mathsf{OR}$. We make this observation explicit in Section 2. Now, a natural question arises: can one extend this idea to total MCSP and prove MCSP $\notin$ P assuming ETH? Ilango suggested that

> *"the most promising approach is to skip* MCSP* *entirely and extend our techniques to apply to* MCSP *directly."* - Rahul Ilango, SIAM J. of Computing, 2022

However, optimal circuits for OR are so well-structured that deciding whether a *total* function is a simple extension of it is actually easy (see the discussion in Section 1.2.2 in [18]). Minimal OR circuits are *read-once formulas*: each of the input is read exactly once and each internal gate has fan-out 1. Simple extensions of it will also be computed by read-once formulas, and deciding whether a given Boolean function has a read-once formula is *easy* [2, 13]. Therefore,

> *"the missing component in extending our results to* MCSP *is finding some function f whose optimal circuits we can characterize but are also sufficiently complex."* - Rahul Ilango, SIAM J. of Computing, 2022

Could simply replacing OR with some read-many f — perhaps XOR, which enjoys tight bounds and a full characterization — allow Ilango's technique to prove MCSP is ETH-hard? For XOR, we show the answer is a resounding **no.** For other potential functions, the answer is more ambiguous. We will discuss prospects for alternative hard functions in Section 1.3.

1.1 Our Results and Contributions

We narrow the field of candidate functions for such a hardness proof by developing a fixed-parameter tractable algorithm for the f-Simple Extension problem (Section D.4). Such an algorithm is surprising, because f-Simple Extension is a meta-complexity problem about *general circuits* and our algorithm works in regimes where we *know* explicit circuit lower bounds. Often, the combinatorial facts used in lower bounds imply a hardness result for the appropriately-restricted meta-complexity problem (e.g., DNF-MCSP)! Nonetheless, we obtain:

▶ **Main Result.** *The f-Simple Extension Problem is in* P *whenever*

1. *$CC(f)$ — the circuit-size complexity of f — is linear,*
2. *the maximum fan-out over all optimal circuits for f is constant, and*
3. *the optimal[4] circuits for f, up to isomorphism and permutation of its n inputs, are efficiently enumerable and polynomial few with respect to the length of its truth table: 2^n.*

[4] In $\mathcal{D}$, we require a circuit to be *normalized* for it to be optimal. In particular, it cannot contain any double-negations. This prevents every function from having an infinite number of optimal circuits.

To apply our main result and discount a particular f, we require an exact specification of its optimal circuits. The next natural candidate — XOR, a simple function whose circuits are neither read-once nor monotone — has been well studied. Beyond enjoying exact size bounds [39, 35], XOR is one of the few functions whose structure has been studied; it is known that, in $\mathcal{R}$, *all* optimal XOR_n circuits are binary trees of $n-1$ XOR_2 sub-blocks [25]. We extend this structural analysis to $\mathcal{D}$ in Section E, obtaining

▶ **Main Lemma** ([25], Theorem 37). *Optimal* XOR_n *circuits consist of* $(n-1)$ $(\neg)\mathsf{XOR}_2$ *sub-circuits.*

Each XOR_n circuit can therefore be characterized using binary trees with n leaves, of which there are $C_{n-1} = O(2^n)$, where C_n is the n^{th} Catalan number [42]. As there are a finite number of optimal normalized $(\neg)\mathsf{XOR}_2$ circuits, combining this characterization and our main result to immediately yields

▶ **Main Corollary.** *The* XOR*-Simple Extension Problem is in* P*.*

Applying Ilango's technique successfully will itself require a deeper study of circuit minimization. This is not merely because any hardness proof needs to bypass our algorithm; knowledge of circuit lower-bounds and optimal constructions for the base function is intrinsic to the reduction itself. We make this connection explicit in Section 2, identifying that

▶ **Main Observation.** f-SEP^* *is ETH-hard under Levin reductions.*

Lastly, in Section 1.3, we inspect each explicit function f that enjoys DeMorgan circuit lower bounds and argue how plausible it is that the optimal set of f circuits avoids our Main Result — a *roadmap* towards ETH-hardness of total MCSP via f-SEP.

1.2 Related Work

(Non-)hardness of MCSP **Variants.** Hirahara showed that *Partial* MCSP is unconditionally NP-hard under *randomized reductions* [14]. Extending his breakthrough result is another approach towards hardness of total MCSP. Though promising, this also faces challenges: under believable cryptographic conjectures (indistinguishability obfuscation and subexponentially-secure one-way functions), GapMCSP is not NP-complete under randomized Levin-reductions [32]. Such reductions appear to suffice for Hirahara's proofs, so one may need new ideas to obtain NP-hardness of total MCSP via his approach.

We bypass the issue by working towards hardness of total MCSP under ETH — a stronger assumption than $\mathsf{P} \neq \mathsf{NP}$. Even so, ETH-hardness of total MCSP remains a major open problem, and there are no known barriers to extending Ilango's approach in this setting. The reader can decide for themselves if our algorithm constitutes such a barrier or not.

Optimal Circuit Structures. Knowing the lower-bounds of some explicit Boolean functions f, a natural question to ask is: *What about the* structure *of every optimal circuit computing* f*?* For some functions and bases, this is easy to answer: minimal Red'kin circuits for OR_n are binary trees of $\vee$ gates. For even slightly more complex functions, structural characterization seems to require intricate and exhaustive case analysis. Some of the earliest work on this question include [38] and [5] which investigated when optimal circuits for certain 2-output Boolean functions must compute each output independently. More recently, Kombarov extended the characterization of XOR circuits to other complete bases when NOT-gates are counted [26].

1.3 Discussion and Future Directions

A Remark on Bases. Both $\mathcal{R}$ and $\mathcal{D}$ are compatible with Ilango's proof of ETH-hardness of MCSP*. That proof relied on functions whose optimal $\{\wedge, \vee, \neg\}$-circuits are read-once monotone formulas. There it was irrelevant whether $\neg$ gates contribute to size: those circuits simply did not contain negations. However, when we move to more complex functions like XOR, negations *must* appear in the optimal circuits and as such, we must decide how to treat them.

In the search for non-linear circuit lower bounds, the choice between μ_R and μ_D is largely irrelevant: the two complexity measures the same up to a small constant factor. However, as we discuss in Section 2, Ilango's technique requires knowledge of the *structure* of optimal circuits. In this setting, there is not necessarily as strong connection between the two bases. By extending Kombarov's characterization of XOR circuits in $\mathcal{R}$ to $\mathcal{D}$ in Section E, we show for that *particular* function, optimal circuits are structurally similar in both bases. But, for other functions, this may well not be the case. It seems likely that negations could enable a function's optimal circuits to greatly vary under the two complexity measures. Indeed, negations can greatly increase a problems complexity: [4] showed that a Boolean function learning problem became difficult only once the number of $\neg$ gates exceeded a small threshold.

By considering both bases in this work, we limit how negations impact the complexity of f-SEP. We show that they do not greatly increase the complexity of the simple extensions themselves: in Section C, we find that simple extensions under both complexity measures are highly structured. If a future hardness reduction relies on negations, their role will be in increasing the structural complexity of the underlying base function, either by increasing the number of distinct optimal circuits, or by enabling non-constant fan-out in a base circuit.

A Roadmap for ETH-Hardness Proofs via Simple Extensions To prove f-SEP is ETH-hard we will need a Boolean function whose optimal circuits are more complex and/or varied than XOR. Specifically, these circuits must either (1) be superlinear in size, (2) require non-constant fanout, or (3) be sufficiently numerous. Superlinear bounds seem beyond current techniques—the most fruitful of which, gate elimination, seems unlikely to be able to prove lower bounds above even $11n$ for wide classes of functions [11]. As such, it seems more sensible to identify Boolean functions which violate the latter conditions. However, structural characterization of the optimal circuits is also hard, and seems to require very tight circuit bounds. Indeed, our DeMorgan basis characterization of XOR repeatedly exploited Schnorr's *exact* $3(n-1)$ bound for XOR_n [39]. Regardless, this greatly narrows the prospective class of functions from the original specification: "more complex than read-once formulas." However, the known explicit functions with tight DeMorgan bounds that may violate these conditions are few and far between. In Table 1[5], we summarize these explicit functions and assess how suitable they are for ETH-hardness of f-Simple Extension Problem.

Observe that every function besides XOR in the table has a (small) gap between the circuit lower and upper bound, and the "$\Omega(1)$ Fanout" column ends with a question mark (?). This is because XOR is the *only* listed function for which we know the *exact* circuit complexity and an optimal circuit characterization. The other "$\Omega(1)$ Fanout" entries above are extrapolated by assuming that their respective DeMorgan *upper bound* constructions

[5] Some of listed bounds are in $\mathcal{U}_2$, the basis consisting of every binary Boolean function besides XOR_2 and $\neg\mathsf{XOR}_2$. For non-degenerate functions besides $f(x) = \neg x$, $\mathcal{U}_2$ and $\mathcal{D}$ are equivalent in terms of size. The multiplexer lower bound of [33] is for the $\mathcal{B}_2$, the basis of all binary Boolean functions, but it also serves as the best known lower bound in $\mathcal{D}$.

Table 1 Explicit Functions with Circuit Lower Bounds in the DeMorgan Basis

Function(s)	Lower Bound		Upper Bound	$\Omega(1)$ Fanout	Source(s)
XOR	$\mathbf{3(n-1)}$	$=$	$\mathbf{3(n-1)}$	NO	[39]
Sum Mod 4	$4n - O(1)$		$5n - O(1)$	NO?	[44]
Sum Mod 2^k	$4n - O(1)$		$7n - o(n)$	NO?	[44]
Multiplexer	$2(n-1)$		$2n + O(\sqrt{n})$	YES?	[34, 24]
Well-Mixed	$5n - o(n)$		poly	MAYBE?	[27, 21]
Weighted Sum of Parities	$5n - o(n)$		$5n + o(n)$	YES?	[1]

are optimal. For instance, Zwick conjectured that optimal circuits computing the Sum Mod 4 (MOD_4) function are "shaped like" ternary full-adder blocks [44]. If this conjecture is true, then MOD_4-Simple Extension can be solved in poly-time since such circuits satisfy the properties of our Main Lemma. Since the MOD_{2^k} functions are computed similarly, we conjecture that exactly characterizing the optimal circuits for Zwick's functions would yield efficient Simple Extension Solvers — not a proof of ETH-hardness for total MCSP.

However, we do have linear lower bounds for functions whose best known constructions have non-constant fanout: the multiplexing function (MUX) contains sub-circuits which are reused a logarithmic number of times [24]. In contrast to XOR however, the bounds for MUX are not tight. The best lower bound is $2(n-1)$, given by Paul [33].

Future Directions. The most obvious next step is to either (1) obtain total characterization of the Multiplexer or (2) extend our Simple Extension Solver to handle circuits with super-constant fanout. Neither of these tasks seems easy, but also they have not been subject to intensive research the way that super-linear circuit lower bounds and hardness of MCSP have. We hope that connecting these kinds of results to ETH-hardness of MCSP provides new perspective and motivation.

Ilango's MCSP^* result has also formed the basis for several hardness results in other models of computation such as formulas and branching programs [19, 9, 10]; could one show that the simple extension problem for formulas or branching programs is hard? These other models of computations may prove easier to work with than unrestricted circuits. They also enjoy superlinear lower bounds thereby bypassing our algorithm. Investigating the simple extension problem in these settings would provide insight into the feasibility of the approach in the circuit setting.

We conclude this section with a discussion of the simple extension problem in general. Despite having only been studied as a tool for proving hardness of MCSP thus far, it may be of independent interest. For example, while hardness of time-bounded Kolmogorov complexity is tightly connected to the existence of one-way functions, hardness of MCSP has much weaker quantitative connections [28, 36]. The f-Simple Extension Problem is more "structured" than MCSP and easily reduces to it, so hardness assumptions about f-Simple Extension Problem are stronger. Could such assumptions imply one-way functions?

1.4 Proof Techniques

1.4.1 The Structure of Optimal XOR Circuits

Similar to [25], we show that *every* optimal $(\neg)\mathsf{XOR}_n$ circuit over the DeMorgan basis partitions into trees of $(n-1)$ sub-circuits computing $(\neg)\mathsf{XOR}_2$ — even when NOT gates are free. The structure of optimal circuits computing the XOR-function is a crucial ingredient for

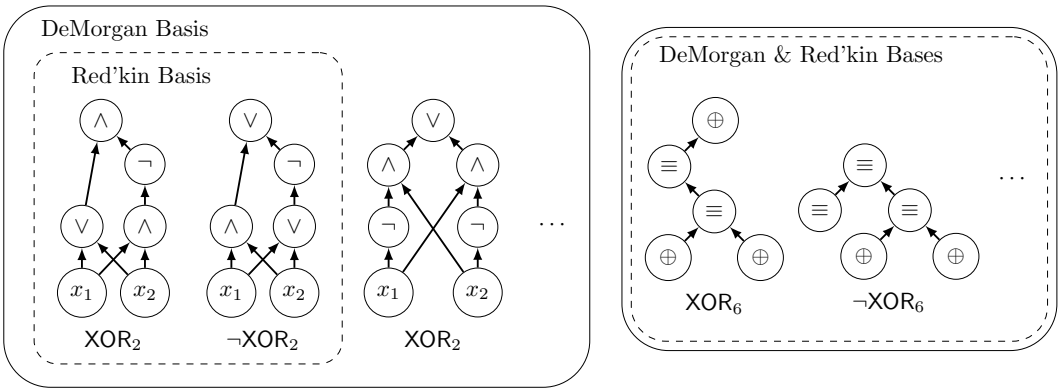


Figure 1 An example of the binary tree structure of optimal circuits computing XOR$_6$. The left sub-figure depicts possible (¬)XOR$_2$ blocks in the Red'kin and DeMorgan Bases. Notice each optimal Red'kin circuit is an optimal DeMorgan circuit, but not vice-versa. The right sub-figure depicts that the arrangement of XOR$_2$ blocks that make up XOR$_6$ circuits are shared by both bases.

ruling it out as a candidate function. We carry out an elementary but intricate case analysis of restricting and eliminating gates from optimal XOR circuits. Essentially we extract more information from the proof of Schnorr's lower bound by using it to identify "templates" that must be found in *any* optimal XOR circuit. We push this process to the limit, fully characterizing the "shape" of all such circuits. Specifically,

- Schnorr's proof is essentially a technical lemma which says that any one-bit restriction will eliminate at least 3 costly gates [39]. This means that at the bottom level of every optimal XOR circuit, any variable must be fed into two distinct costly gates, and furthermore, one of these two must be fed into another costly gate. Any deviation from these properties will violate essential properties of the XOR-function, such as "XOR depends on all the input bits." Via a basic inductive argument and the fact that XOR is downward self-reducible, Schnorr's lower bound follows: $CC(\mathsf{XOR}_n) \geq 3(n-1)$.
- Schnorr's proof leaves the local structure of the optimal circuit computing XOR "open." Namely, it does not provide any information about the other inputs of the costly gates or where their outputs connect to the rest of the circuit, since we consider fan-in 2 and unbounded fan-out. However, we know that XOR circuit has a matching upper-bound of $3(n-1)$. In particular, this means *each one-bit restriction cannot remove more than* 3 *gates*. We also know that *each variable in optimal* XOR*-circuits must be read twice.*
- We leverage these two properties to show that in every optimal XOR circuit, any two distinct input variables x_i and x_j must be fed into a block $\mathcal{B}$ as shown in the left sub-figure of Figure 1. Specifically, we argue that any deviations from the block will violate at least one of the properties via exhaustive case analyses of gate elimination steps. Finally, we argue that this block $\mathcal{B}$ must compute either XOR$_2$ or ¬XOR$_2$ and apply a basic inductive argument to obtain the desired structural characterization of any optimal circuit computing XOR$_n$ as depicted in the right sub-figure of Figure 1.

Besides the linear size for optimal circuits computing XOR, our structural theorem yields two more properties that rule out XOR as a candidate function for MCSP-hardness via Simple Extension. That is, for optimal circuits computing XOR$_n$, (1) *the maximum fan-out is a constant*, and (2) *the number of such optimal circuits up to permutation of variables, is* $2^{O(n)}$.

Algorithm 1 Informal Simple Extension Solver, taking input $n \in \mathbb{N}$, $g \in \mathcal{F}_{n+m}$

if there is no key to f in g or g is degenerate **then**
    return False
▷ If the above tests pass, then g is a non-degenerate extension of f. It remains to check simplicity.
for each isomorphism class $\mathcal{C}$ of open optimal circuits for f **do**
    $F \leftarrow$ an arbitrary element of $\mathcal{C}$ with all gates labelled in topological order
    label the open nodes of F by an arbitrary permutation of $x_1, \ldots, x_n$
    for each reverse elimination E that adds exactly m costly gates to F **do**
        $\tilde{G} \leftarrow \texttt{Decode}(F, E)$
        if $\texttt{tt}(\tilde{G}) \simeq \texttt{tt}(g)$ **then** ▷ Test using the procedure of Theorem 26.
            return True
return False

1.4.2 A Fixed-Parameter Tractable Simple Extension Solver

It is easy to see that the approach of solving the Simple Extension Problem for XOR_n via brute-forcing over all possible circuits of size $CC(f) + m$ is super-polynomial in terms of the length of the input truth-tables. Using the following ingredients, we design Algorithm 1 below, a Fixed-Parameter Tractable (FPT) algorithm for the Simple Extension problem that depends on the following three parameters: (1) the number of optimal circuits for f (up to isomorphism & permutation of variables), (2) the maximum fanout of any node in any optimal circuit for f, and (3) $CC(f)$.

Structured Simple Extension Circuits. By analyzing the *behavior* of optimal simple extension circuits under gate elimination, we are able to characterize the *structure* of *every* optimal circuit computing a simple extension. By definition, if g is a simple extension of f then there are restrictions of g's added variables (called *extension* variables and denoted y_i) that yield f. We call such a restriction a *key* to f in g. We first show that *circuits obtained by partially restricting with a key are themselves optimal simple extension circuits for intermediate extensions*. Building on this, we then develop convenient *all-stops restrictions* that order substitutions and simplification steps with the following properties: (1) *single-bit substitutions from this key in the given order eliminate exactly one costly gate at each step*, (2) *there exists such an all-stops restriction for any optimal circuit computing a simple extension* (Lemma 17).

Combining these tools, we inductively show a robust structure arises in optimal simple extension circuits: *each extension variable occurs in an isolated read-once subformula that depends only on other extension variables* (referred to to as the *Y-trees*). Formally,

▶ **Definition 1** (Y-Tree Decomposition). *Let G be a circuit with two distinguished sets of inputs:* base variables X *and* extension variables Y. *A Y-*Tree Decomposition *of G is a set of triples $\langle \gamma, b, T \rangle$ where γ—referred to as a* combiner*—is a costly gate of G, bit $b \in \{0, 1\}$ designates an input of γ, and T is a sub-circuit of G rooted at the b child of γ such that*

1. *Each T is a read-once formula in only extension variables Y.*
2. *Each $y_i \in Y$ appears in at most one T.*
3. *Each T is* isolated *in G — gate γ is the unique gate reading from T, and it only reads the root of T.*
4. *The sub-circuit of G rooted at the $\neg b$ child of γ contains at least one X variable.*

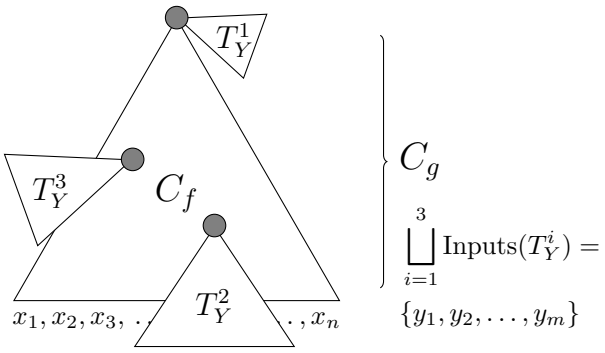


Figure 2 An example of a Y-Tree Decomposition of size three.

The size *of a decomposition is the number of tuples — Y-trees and their associated combiner gates — present in the decomposition. The* weight *of a Y-tree decomposition is the number of extension variables that are read in some T. We say a Y-tree decomposition is* total *if its weight is $|Y|$, i.e. every extension variable appears. An example Y-tree decomposition of size three is depicted in Figure 2, where the shaded circles represent the circuity around each combiner γ connecting each $T_{\mathcal{Y}}$ to the rest of the circuit.*

When gate elimination is performed with a total key, these added Y-trees and their combiners are pruned to reveal an embedded optimal circuit for the base f function. We get the following structural insight: *every optimal simple extension circuit has a total Y-tree decomposition.* This decomposition forms the basis of our strategy: we brute force over every optimal base circuit and try to "splice in" every possible Y-tree.

Encoding & Decoding the "Grafts" in a Y-Tree Decomposition. To ensure we can efficiently construct these candidate simple extension circuits, we devise an encoding scheme and corresponding decoding algorithm which efficiently captures the difference in local neighborhoods after each new Y-tree is spliced on top of an existing gate or input. Our final encoding must be $O(n+m)$ bits long to ensure brute-force runs in $2^{O(n+m)}$. We present our encoding as a communication problem to clarify the overhead and constraints involved.

Suppose g is a simple extension of f and Alice knows G, an optimal circuit for g. Alice can obtain an optimal circuit F computing f by simply restricting the y-variables of G with a key and performing gate elimination. Now consider the following communication problem: Bob (i.e., line 5 of Algorithm 1) knows F, and Alice would like to send him G using as few bits as possible. Because g is a simple extension of f, Alice can compute the Y-tree decomposition of optimal circuit G. The idea is to send Bob a sequence of instructions that tell him exactly how to graft each Y-Tree of G onto the gates of F, where all information is encoded relative to isomorphism-invariant properties of F.

Speeding Up Via Truth-table Isomorphism. Brute-forcing over the total encodings described above is still incredibly inefficient. Since each Y-tree is a read-once formula in the added m variables there are least $C_{m-1} \cdot m!$ such explicit Y-trees, where C_a is the a^{th} Catalan number [42]. The dominating term—$m!$—comes from permuting the labels of the variables. The same issue arises if our base function f is symmetric: the number of optimal f circuits is $\Omega(n!)$.

We sidestep this issue and drastically improve the speed of brute-force search. If we "incorrectly" assign variables in the base circuit or in the Y-trees, the result is a circuit for

a Boolean function that is *truth-table isomorphic* to g, i.e. their truth-tables are the same up to a permutation of the inputs. If h and g are truth-table isomorphic then they have the same circuit complexity. Thus it suffices to brute-force over unlabeled ("open") base circuits and Y-trees, assign variables to inputs arbitrarily, generate each circuit's truth table, and check if it is truth-table isomorphic to g. This final step is feasible since truth-table isomorphism testing can be done in polynomial time [30].

This results in our final algorithm, which runs in time $O(|L| \cdot 2^{O(\ell(s+m))})$ where $|L|$ is the number of optimal base f circuits up to permutation of variables, ℓ is the maximum fanout in any of those base circuits, and $s = CC(f)$. As discussed above, for XOR these parameters are all sufficiently small and hence XOR-Simple Extension is in P.

1.4.3 Paper Outline

The paper is laid out as follows. Section 2 explores the implicit reduction present to f-SEP in the ETH-hardness proof for MCSP* in [18]. This discussion yields our Main Observation and motivates our study of f-SEP and circuit structures. It, in conjunction with this introduction, constitutes an extended abstract of our paper.

Appendix A and Appendix B formally define circuits and gate elimination respectively. In Appendix C, we establish that every optimal circuit computing simple extensions are highly structured: they can be decomposed into Y-trees. This observation forms the basis of our Main Result, a fixed parameter tractable algorithm for f-SEP, in Appendix D. Finally, in Appendix E, we extend Kombarov's characterization of optimal XOR circuits in $\mathcal{R}$ to $\mathcal{D}$, in order to apply our Main Theorem to XOR-SEP.

The dependence between sections varies. For example, Appendix E is independent of the f-SEP material like Section 2 and Appendix C. To aid the reader, we provide a reading order in Figure 3.

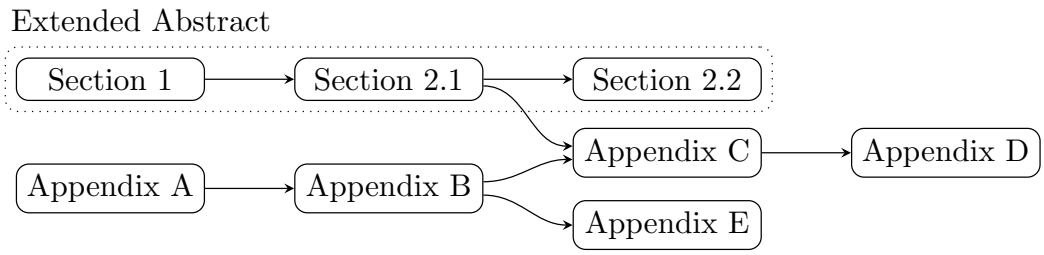


Figure 3 The structure of our paper. Sections 1 and 2 form an extended abstract. An incoming arrow indicates that the material depends on the previous section.

2 Revisiting ETH hardness for MCSP* via Simple Extensions

We re-examine the proof that MCSP* is ETH-hard from [18]. Our aim is to better understand the reduction from $2n \times 2n$ Bipartite Permutation Independent Set (BPIS) to the Partial f Simple Extension problem (f-SEP*).

2.1 The f-Simple Extension Problem

We give formal definitions of simple extensions and its associated decision problem. We first define non-degeneracy of a Boolean function.

▶ **Definition 2.** *A function $f \in \mathcal{F}_n$, the set of Boolean functions on n variables, depends on its i^{th} variable, x_i, if there exists an input $\alpha \in \mathcal{F}_n$ such that $f(\alpha) \neq f(\alpha \oplus e_i)$, where e_i denotes the Boolean vector that is 0 everywhere except for a 1 at index i and $\oplus$ is bitwise* XOR. *If f depends on all of its variables, then we say f is a* non-degenerate *function. Conversely, we say f is a* degenerate *function if it does not depend on at least one variable.*

We now define simple extension as

▶ **Definition 3.** *Let $f \in \mathcal{F}_n$ be non-degenerate. A* simple extension *of f is either f itself or a function $g \in \mathcal{F}_{n+m}$ satisfying:*

1. *g is a non-degenerate function,*
2. *$CC(g) = CC(f) + m$, and*
3. *there exists a setting $k \in \{0,1\}^m$, called a* key, *such that for all $x \in \{0,1\}^n$, $g(x,k) = f(x)$.*

We denote the first n inputs of f and g by $x_1, \ldots x_n$ and will refer to the extra m inputs of g as *extension variables* and refer to them as $y_1, \ldots, y_m$. From the definition of simple extension above, we define the following decision problem.

▶ **Problem 1** (f-SEP). *Let f be a sequence of Boolean functions $\{f_n\}_{n\in\mathbb{N}}$ such that each f_n is a non-degenerate function in $\mathcal{F}_n$. The f-Simple Extension Problem is defined as follows: Given $n \in \mathbb{N}$ and $tt(g)$—the truth tables of a binary function $g \in \mathcal{F}_{n+m}$—decide whether g is a simple extension of f_n.*

We extend this to partial functions,

▶ **Problem 2** (f-SEP*). *Given $n \in \mathbb{N}$ and a* partial *$tt(g)$, decide whether any completion of the truth table is a simple extension of f_n.*

2.2 An Explicit Reduction BPIS from to f-SEP*

Recall the formulation of BPIS from [18],

▶ **Problem 3** (BPIS). *The $2n \times 2n$ Bipartite Permutation Independent Set is defined as follows: Given $G = (V, E)$ a directed graph with vertex set $V = [n] \times [n]$. Decide whether there exists a permutation $\pi : [2n] \to [2n]$ such that:*

1. *$\pi([n]) = [n]$,*
2. *$\pi(\{n + i : i \in [n]\}) = \{n + i : i \in [n]\}$,*
3. *if $((j,k),(j',k')) \in E$, then either $\pi(j) \neq k$ or $\pi(j' + n) \neq k' + n$*

If the Exponential Time Hypothesis holds, then BPIS cannot be solved much faster than brute-forcing over all $2^{o(n \log n)}$ permutations [29]. BPIS is a natural problem to show hardness of circuit size problems since there are $O(2^{s \log s})$ circuits of size s. Reducing from BPIS implies, assuming ETH, that our problem can not be solved much faster than by brute-forcing over all possible circuits.

The reduction from BPIS to f-SEP*, as it appears as part of the original proof, is somewhat *implicit*; the target problem, as written, could be described more simply as "determine whether a partial truth table is ever consistent with a monotone read-once formula." Since this is *easy* for total functions [2, 13], this view of the reduction cannot help us to extend the reduction technique to total MCSP. We will need the more general f-SEP and by reframing Ilango's proof we gain insight into how it might extend to total functions.

The connection to f-SEP* is *explicitly* stated, however, in the introduction of [18]. Here, the reduction is identified with OR$_{4n}$-SEP* where each z variable is an extension variable.

This is true, though non-degenerate functions computed by read-once formulas are simple extensions of *any* of their non-degenerate restrictions. When framing the reduction to f-SEP* explicitly, we find it more compelling to choose a different function $\hat{f}$, whose truth table is given by $\bigvee_{i\in[2n]}(y_i \wedge z_i)$ — because using $\hat{f}$ makes the intuitive description of Ilango's technique "reverse gate elimination" an obvious property of the reduction. Under this framing, the extension variables will instead be the x variables in Ilango's original proof. Despite this, fixing f to be OR_{4n} is *not* arbitrary; afterwards, we discuss what insight it provides. Informally, the two choices of base function provide distinct "channels" for ETH to imply hardness of f-SEP*.

Structural Lemmas To encode BPIS in $\hat{f}$-SEP*, we first prove two lemmas which were essentially proven in tandem in [18] as Lemma 16. We separate them out here and stay faithful to the original arguments. Like Lemma 16, these lemmas establish structural properties of circuits computing our base function $\hat{f}$ and our eventual output $\hat{g}$.

▶ **Lemma 4.** *Let $\hat{f} : \{0,1\}^{2n} \times \{0,1\}^{2n} \to \{0,1\}$ be the Boolean function computed by $\bigvee_{i\in[2n]}(y_i \wedge z_i)$. If ψ be an optimal normalized*[6] *formula computing $\hat{f}$, then ψ, as a formula, is equal to $\bigvee_{i\in[2n]}(y_i \wedge z_i)$.*

In particular, the only difference between two distinct optimal circuits for $\hat{f}$ is which binary tree of fanin 2 $\vee$ gates is used.

Proof. Observe that ψ must read all of its input variables and furthermore must do so positively. This is because (1) f depends on all of its variables and is monotone in them, and (2) ψ is normalized and thus no $\neg$ gates can appear internally in the circuit. We note that ψ must use a total of $4n-1$ gates since f is non-degenerate on $4n$ variables computable. We see that ψ must contain at least $2n-1$ $\vee$ gates: substituting $z = 1^{2n}$ and simplifying yields a monotone read once formula computing $\mathsf{OR}_{2n}(y_1, \ldots, y_{2n})$ which must consist of $2n-1$ $\vee$ gates that appear in ψ.

We now argue that each z_i feeds into an $\wedge$ gate. Assume otherwise, then observe that setting $z_i = 1$ and simplifying removes at least two gates (since $z_1 \vee p \equiv 1 \vee p \equiv 1$). The resulting read once formula for $\hat{f}|_{z_i \leftarrow 1}$ uses at most $4n-3$ gates. This is a contradiction since $\hat{f}|_{z_i \leftarrow 1}$ still depends on all of its remaining $4n-1$ variables: it cannot be computed by a circuit with fewer than $4n-2$ gates.

We now argue the other input to the $\wedge$ gate fed by z_i is y_i. Assume otherwise. Notice that since ψ is read-once, setting $z_i \leftarrow 0$ simplifying disconnects the other input to $\wedge$ gate and thus removes dependence on any variables it depends on. However, $\hat{f}|_{z_i \leftarrow 0}$ still depends on all of its variables besides y_i. Thus the $\wedge$ gate can only read y_i.

Since each z_i and y_i feed a distinct $\wedge$ gate, there are at least $2n$ $\wedge$ gates. Since there are $4n-1$ total gates, and at least $2n-1$ $\vee$ gates, we know that these $\wedge$ gates are the only $\wedge$ gates that appear. Thus the remainder of the circuit is a binary tree of $2n-1$ $\vee$ gates whose $2n$ leaves are the $2n$ $\wedge$ gates. ◀

Having established the structure of optimal circuits computing $\hat{f}$ we can further restrict its simple extensions. Extra restrictions to the truth table enforce that extension variables must be spliced into the circuit using $2n$ additional $\vee$ gates that each read a different y_i.

[6] A formula is normalized if all negations are pushed down to the input level. Normalization does not affect the size of the formula, and thus f-SEP* still reduces to MCSP* even if we restrict ourself to normalized formulas.

Table 2 BPIS requirements and the corresponding restrictions on $\hat{g}$

BPIS Requirement on σ	Corresponding $\hat{g}$ Restriction	Impact If π Violates
$\sigma(\{1,\dots,n\}) = \{1,\dots,n\}$	$\mathsf{OR}_n(x_1,\dots,x_n)$ when $z = 1^n0^n$ and $y = 0^{2n}$	If $\pi(i) = j \geq n$, then $z_j \leftarrow 0$ removes x_i in C_π
$\sigma(\{n+1,\dots,2n\}) = \{n+1,\dots,2n\}$	$\mathsf{OR}_n(x_{n+1},\dots x_{2n})$ when $z = 0^n1^n$ and $y = 0^{2n}$	As above, $C_\pi\restriction_{z_j \leftarrow 0}$ will not depend on x_i
If $((j,k),(j',k')) \in E$ then $\sigma(j) \neq k$ or $\sigma(n+j') \neq \sigma(n+k')$	1 if $(x,y,z) = (\overline{e_k e_{k'}}, 0^{2n}, e_j e_{j'})$ where $\exists((j,k),(j',k')) \in E$	C_π wrongly outputs 0

This pairing of each y_i with a different x_j will define a permutation in which we can encode BPIS solutions.

▶ **Lemma 5.** *Let $g : \{0,1\}^{2n} \times \{0,1\}^{2n} \times \{0,1\}^{2n} \to \{0,1\}$ be any simple extension of $\hat{f}$ satisfying the following conditions:*

$$g(x,y,z) = \begin{cases} \hat{f}(y,z) & \text{if } x = 0^{2n} \\ \mathsf{OR}_{2n}(z_1,\dots z_{2n}) & \text{if } x = 1^{2n} \\ \mathsf{OR}_{4n}(x_1,\dots x_{2n}, y_1,\dots y_{2n}) & \text{if } z = 1^{2n} \\ 0 & \text{if } z = 0^{2n} \end{cases}$$

If ϕ is an optimal normalized formula computing g then there exists a permutation $\pi : [2n] \to [2n]$ such that ϕ equals, as a formula, $\bigvee_{i\in[2n]}((x_{\pi(i)} \vee y_i) \wedge z_i)$.

Proof. Since $g(x,y,z) = \hat{f}(y,z)$ when $x = 0^{2n}$, we know that ϕ must read all y and z variables positively. Similarly, all x variables must be read positively, since $g(x,y,1^{2n}) = \mathsf{OR}_{4n}(x_1,\dots x_{2n}, y_1,\dots y_{2n})$. Note that these restrictions also imply that ϕ contains exactly $4n-1$ $\vee$ gates and $2n$ $\wedge$ gates since substituting and simplifying yields optimal formulas for those restrictions. From the lemma above, we know know that when setting $x = 0^{2n}$ and simplifying, we obtain a circuit structurally equivalent to $\bigvee_{i\in[2n]}(y_i \wedge z_i)$. Therefore $2n$ $\vee$ gates must be removed during simplification. These $\vee$ gates cannot feed any remaining $\vee$ above the $\wedge$ gates, since otherwise setting $x = 1^{2n}$ would fix the circuit to be 1, rather than $\mathsf{OR}_{2n}(z_1,\dots z_{2n})$. Similarly, z_i cannot feed any of these $\vee$ gates, as setting $x = 1^{2n}$ would remove dependence on that z_i since the circuit is a read-once formula. Thus each $\vee$ gate can only depend on the x and y variables. Observe that each y_i must feed into one of these $\vee$ gates instead of the $\wedge$ gate fed by z_i, as otherwise when we set $x = 1^{2n}$, the function would still depend on y_i. Since there are exactly $2n$ additional $\vee$ gates, and exactly $2n$ y and $2n$ x variables, its easy to see that each additional $\vee$ gate must read one x_i and one y_j. Therefore, as a formula, ϕ must be $\bigvee_{i\in[2n]}((x_{\pi(i)\vee y_i}) \wedge z_i)$ for some permutation π. ◀

An Explicit Reduction We now provide an explicit reduction from BPIS to $\hat{f}$-SEP*. Given an instance G of BPIS we output $4n$ and the partial truth-table for a function $\hat{g}$ that is consistent with the requirements of Lemma 5. We add three additional restrictions (listed in Table 2) to ensure that any permutation π, whose corresponding circuit $C_\pi \equiv \bigvee_{i\in[2n]}\left((x_{\pi(i)} \vee y_i) \wedge z_i\right)$ is consistent with $\hat{g}$, *is* also a solution for G (and vice versa). All other rows of the truth table are left undefined (e.g. as $\star$). We summarize the requirements for any valid BPIS solution σ, the corresponding restriction, and how it enforces the requirement on π in Table 2.

This completes the reduction as any σ satisfying BPIS for G can be used to construct a read-once circuit consistent with $\hat{g}$ and vice versa. The arguments verifying this are the same as in [18], and we refer the reader there for the full details.

▶ **Lemma 6.** BPIS *reduces to* $\hat{f}$-SEP* *in* $2^{O(n)}$ *time.*

The Original Framing In its introduction, [18] identifies $\hat{g}$ as a simple extension of OR_{4n}. Under this lens, the hardness comes from determining which OR_{4n} base circuit can have the z extension variables added. The additional truth-table restrictions on $\hat{g}$ force each z_i to be spliced in a particular way adjacent to y_i. Assuming ETH, there are $\Omega(n!)$ optimal base OR_{4n} circuits that must be checked via brute-force.

Implicit Circuit Lower Bounds & Enumeration of Optimal Circuits From both presentations, we see that leveraging f-SEP involves explicit circuit size lower bounds. Indeed, both Lemma 4 and Lemma 5 prove formula lower bounds for specific non-degenerate functions. However, in a sense, circuit lower bounds are intrinsic to the reduction itself. This connection can be made rigorous: the reduction can used to produce explicit Boolean functions which enjoy non-vacuous lower bounds. On `no` instance of BPIS, the reduction outputs a partial truth table where every completion is non-degenerate but *not* a simple extension. Hence, the circuit complexity of these completions is not the vacuous $6n - 1$ lower bound obtained by knowing that functions produced are non-degenerate.

Furthermore, the reduction did not solely rely on the circuit complexity of $\hat{f}$ and $\hat{g}$. Lemmas 4 and 5 *tightly control* how base circuits and their extensions can be arranged; and this is pivotal for encoding BPIS permutations. This structural requirement can be formalized by observing the reduction is also an efficient *Levin reduction.*

Recall, from [32], that a Levin reduction is a many-one reduction that also efficiently maps witnesses, not just problem instances. More precisely, let R be a set of ordered pairs (x, w) where x is a yes-instance of a problem and w is an accompanying certificate. We define L_R, the language defined by R, to be the set of elements x such that $(x, w) \in R$ for some w. Then a Levin reduction between two languages A and B is an efficient many-one reduction r between problem instances paired with two efficient mappings m, ℓ between instance-witness pairs that satisfy (1) if $(x, w) \in R_A$ then $(r(x), m(x, w)) \in R_B$ and (2) $(t(x), w) \in R_B$ implies $(x, \ell(x, w)) \in R_A$.

For BPIS, the witnesses for an instance are simply the valid permutations σ and witnesses for $\hat{f}$-SEP are optimal circuits computing the extension. Let $\mathcal{R}_{\mathsf{BPIS}}$ and $\mathcal{R}_{\hat{f}\text{-SEP}^*}$ be the sets of ordered pairs consisting of problem instances and all of their witnesses as described. The reduction admits linear time mappings between witnesses: given σ, simply construct $\bigvee_{i\in[2n]}((x_{\sigma(i)} \vee y_i) \wedge z_i)$ and given a circuit for $\hat{g}$, simply read off the permutation from the x variables.

References

1. Kazuyuki Amano and Jun Tarui. A well-mixed function with circuit complexity 5n: Tightness of the lachish–raz-type bounds. *Theoretical computer science*, 412(18):1646–1651, 2011.
2. Dana Angluin, Lisa Hellerstein, and Marek Karpinski. Learning read-once formulas with queries. *J. ACM*, 40(1):185–210, 1993. doi:10.1145/138027.138061.
3. Vikraman Arvind and Yadu Vasudev. Isomorphism testing of boolean functions computable by constant-depth circuits. *Inf. Comput.*, 239:3–12, 2014. URL: https://doi.org/10.1016/j.ic.2014.08.003, doi:10.1016/J.IC.2014.08.003.
4. Eric Blais, Clément L Canonne, Igor C Oliveira, Rocco A Servedio, and Li-Yang Tan. Learning circuits with few negations. *arXiv preprint arXiv:1410.8420*, 2014.
5. Norbert Blum and Martin Seysen. Characterization of all optimal networks for a simultaneous computation of AND and NOR. *Acta Informatica*, 21:171–181, 1984. doi:10.1007/BF00289238.
6. Mahdi Cheraghchi, Valentine Kabanets, Zhenjian Lu, and Dimitrios Myrisiotis. Circuit lower bounds for mcsp from local pseudorandom generators. *ACM Transactions on Computation Theory (TOCT)*, 12(3):1–27, 2020.
7. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.
8. William Feller and Philip M Morse. An introduction to probability theory and its applications. vol i, 1968.
9. Ludmila Glinskih and Artur Riazanov. MCSP is hard for read-once nondeterministic branching programs. In Armando Castañeda and Francisco Rodríguez-Henríquez, editors, *LATIN 2022: Theoretical Informatics - 15th Latin American Symposium, Guanajuato, Mexico, November 7-11, 2022, Proceedings*, volume 13568 of *Lecture Notes in Computer Science*, pages 626–640. Springer, 2022. doi:10.1007/978-3-031-20624-5_38.
10. Ludmila Glinskih and Artur Riazanov. Partial minimum branching program size problem is eth-hard. *Electron. Colloquium Comput. Complex.*, TR24-117, 2024. (To Appear in ITCS 2025). URL: https://eccc.weizmann.ac.il/report/2024/117, arXiv:TR24-117.
11. Alexander Golovnev, Edward A. Hirsch, Alexander Knop, and Alexander S. Kulikov. On the limits of gate elimination. *J. Comput. Syst. Sci.*, 96:107–119, 2018. doi:10.1016/j.jcss.2018.04.005.
12. Alexander Golovnev, Rahul Ilango, Russell Impagliazzo, Valentine Kabanets, Antonina Kolokolova, and Avishay Tal. Ac0 [p] lower bounds against mcsp via the coin problem. In *ICALP*, 2019.
13. Martin Charles Golumbic, Aviad Mintz, and Udi Rotics. Factoring and recognition of read-once functions using cographs and normality and the readability of functions associated with partial k-trees. *Discrete Applied Mathematics*, 154(10):1465–1477, 2006. URL: https://www.sciencedirect.com/science/article/pii/S0166218X06000072, doi:https://doi.org/10.1016/j.dam.2005.09.016.
14. Shuichi Hirahara. Np-hardness of learning programs and partial MCSP. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 968–979. IEEE, 2022. doi:10.1109/FOCS54457.2022.00095.
15. Shuichi Hirahara, Igor C. Oliveira, and Rahul Santhanam. Np-hardness of minimum circuit size problem for OR-AND-MOD circuits. In Rocco A. Servedio, editor, *33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA*, volume 102 of *LIPIcs*, pages 5:1–5:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. URL: https://doi.org/10.4230/LIPIcs.CCC.2018.5, doi:10.4230/LIPICS.CCC.2018.5.
16. Shuichi Hirahara and Rahul Santhanam. On the average-case complexity of mcsp and its variants. In *32nd Computational Complexity Conference (CCC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

17 Yizhi Huang, Rahul Ilango, and Hanlin Ren. Np-hardness of approximating meta-complexity: A cryptographic approach. *Cryptology ePrint Archive*, 2023.

18 Rahul Ilango. Constant depth formula and partial function versions of mcsp are hard. *SIAM Journal on Computing*, 0(0):FOCS20–317–FOCS20–367, 2020. arXiv:https://doi.org/10.1137/20M1383562, doi:10.1137/20M1383562.

19 Rahul Ilango. The minimum formula size problem is (ETH) hard. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 427–432. IEEE, 2021. doi:10.1109/FOCS52979.2021.00050.

20 Rahul Ilango, Bruno Loff, and Igor Carboni Oliveira. Np-hardness of circuit minimization for multi-output functions. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 27, page 21, 2020.

21 Kazuo Iwama and Hiroki Morizumi. An explicit lower bound of 5n - o(n) for boolean circuits. In Krzysztof Diks and Wojciech Rytter, editors, *Mathematical Foundations of Computer Science 2002, 27th International Symposium, MFCS 2002, Warsaw, Poland, August 26-30, 2002, Proceedings*, volume 2420 of *Lecture Notes in Computer Science*, pages 353–364. Springer, 2002. doi:10.1007/3-540-45687-2_29.

22 Stasys Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms and combinatorics*. Springer, 2012. doi:10.1007/978-3-642-24508-4.

23 Valentine Kabanets and Jin-yi Cai. Circuit minimization problem. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 73–79. ACM, 2000. doi:10.1145/335305.335314.

24 Klein and Paterson. Asymptotically optimal circuit for a storage access function. *IEEE Transactions on Computers*, C-29(8):737–738, 1980. doi:10.1109/TC.1980.1675657.

25 Yu A Kombarov. The minimal circuits for linear boolean functions. *Moscow University Mathematics Bulletin*, 66(6):260–263, 2011.

26 Yu A Kombarov. Complexity and structure of circuits for parity functions. *Journal of Mathematical Sciences*, 233:95–99, 2018.

27 Oded Lachish and Ran Raz. Explicit lower bound of 4.5n - o(n) for boolena circuits. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, STOC '01, page 399–408, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/380752.380832.

28 Yanyi Liu and Rafael Pass. On one-way functions and kolmogorov complexity. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1243–1254. IEEE, 2020. doi:10.1109/FOCS46700.2020.00118.

29 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. *SIAM Journal on Computing*, 47(3):675–702, 2018. arXiv:https://doi.org/10.1137/16M1104834, doi:10.1137/16M1104834.

30 Eugene M Luks. Hypergraph isomorphism and structural equivalence of boolean functions. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 652–658, 1999.

31 William J Masek. Some np-complete set covering problems. *Unpublished manuscript*, 1979.

32 Noam Mazor and Rafael Pass. Gap MCSP is not (levin) np-complete in obfustopia. In Rahul Santhanam, editor, *39th Computational Complexity Conference, CCC 2024, July 22-25, 2024, Ann Arbor, MI, USA*, volume 300 of *LIPIcs*, pages 36:1–36:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: https://doi.org/10.4230/LIPIcs.CCC.2024.36, doi:10.4230/LIPICS.CCC.2024.36.

33 Wolfgang J. Paul. A 2.5 n-lower bound on the combinational complexity of boolean functions. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, page 27–36, New York, NY, USA, 1975. Association for Computing Machinery. doi:10.1145/800116.803750.

34 Wolfgang J. Paul. A 2.5 n-lower bound on the combinational complexity of boolean functions. *SIAM J. Comput.*, 6(3):427–443, 1977. doi:10.1137/0206030.

35 NP Red'kin. Proof of minimality of circuits consisting of functional elements. *Systems Theory Research: Problemy Kibernetiki*, pages 85–103, 1973.

36 Hanlin Ren and Rahul Santhanam. Hardness of kt characterizes parallel cryptography. *Cryptology ePrint Archive*, 2021.

37 Rahul Santhanam. Pseudorandomness and the minimum circuit size problem. *LIPIcs*, 151, 2020.

38 Jürgen Sattler. Netzwerke zur simultanen berechnung boolescher funktionen. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 32–40. Springer, 1981. URL: https://doi.org/10.1007/BFb0017293, doi:10.1007/BFB0017293.

39 Claus-Peter Schnorr. Zwei lineare untere schranken für die komplexität boolescher funktionen. *Computing*, 13(2):155–171, 1974. doi:10.1007/BF02246615.

40 Claude. E. Shannon. The synthesis of two-terminal switching circuits. *The Bell System Technical Journal*, 28(1):59–98, 1949. doi:10.1002/j.1538-7305.1949.tb03624.x.

41 B. A. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *Annals of the History of Computing*, 6(4):384–400, 1984. doi:10.1109/MAHC.1984.10036.

42 J. H. van Lint and R. M. Wilson. *Recursions and generating functions*, page 109–1311. Cambridge University Press, 1992.

43 Ingo Wegener. *The Complexity of Boolean Functions*. Wiley-Teubner, 1987.

44 Uri Zwick. A 4n lower bound on the combinational complexity of certain symmetric boolean functions over the basis of unate dyadic boolean functions. *SIAM Journal on Computing*, 20(3):499–505, 1991.

A Circuits

In this section, we precisely define Boolean circuit complexity and accompanying notions related to the "shapes" and "parts" of circuits. These definitions are straightforward, but the details matter because we are working with *general* circuits and functions of *linear* size complexity. We cannot, for example, afford to layer circuits or put them into negation normal form (NNF, all NOT-gates on the inputs) because size-optimal circuits do not (in general) obey these restrictions. In particular, optimal circuits for XOR are neither layered nor in NNF (Theorem 37). We will require normal forms that are *free* to impose (Lemma 10).

Define general circuits over the basis $\mathcal{B} = \{\wedge, \vee, \neg, 0, 1\}$ of Boolean functions: binary $\wedge$ and $\vee$, unary $\neg$ and zero-ary (constants) 1 and 0. Circuits take zero-ary variables in $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots y_m\}$ for some fixed n and m as inputs. Throughout this work, we use the standard formulation of circuits consisting of single-sink and multi-source DAGs where internal nodes, called gates, are labeled by function symbols, sources are labeled by input variables, and edges as "wires" between the gates. We write V_C and E_C to denote the set of nodes and the set of edges of a circuit C respectively, omitting the subscript when it is clear which circuit is being referenced. Circuits compute Boolean functions through *substitution* followed by *evaluation*.

An *assignment* ρ of input variables is a mapping from the set of inputs to $\{0, 1\}$. To substitute into a circuit according to an assignment ρ, each input x_i is replaced by the constant $\rho(x_i)$. For a circuit G and assignment ρ we write $G\restriction_\rho$ to denote the circuit obtained by substituting into G according to ρ. To evaluate a circuit, the values of interior nodes labeled by function symbols are computed in increasing topological order. Each gate's value is obtained by applying it's function to the value of the node's incoming wires. The output of the circuit overall is the value of its sink.

Let $\mathcal{F}_n$ be the family of Boolean functions on n variables. We say a circuit G on n variables computes $g \in \mathcal{F}_n$ if for all $\alpha \in \{0, 1\}^n$, $G(\alpha) = g(\alpha)$. The size of a circuit G is denoted $|G|$ and $CC(g)$, the circuit complexity of a Boolean function g, is the minimum size of any circuit computing g. We study two notions of circuit size, $\mu_{\mathcal{D}}$ and $\mu_{\mathcal{R}}$, which count the number of binary gates and the total number of gates respectively. We decorate any notation with $\mathcal{D}$ or $\mathcal{R}$ (e.g. $CC(G)^{\mathcal{D}}$) to denote μ_D and μ_R whenever necessary, but will omit if the measure is clear from context or if a statement holds for both measures. We say a circuit G computing g is optimal if $|G| = CC(g)$ and, in $\mathcal{D}$, require that the circuit is *normalized*, i.e. does not contain double-negations and every non-$\neg$ node feeds at most one $\neg$ gate.

We will often order the nodes of a circuit by *depth*, the maximum number of binary gates on any path from the node to the output. We sort a circuits nodes in *decreasing* order by depth, i.e. $\texttt{depth}(u) > \texttt{depth}(v)$ implies u appears before v in our ordering, breaking ties arbitrarily. Such orderings are efficiently realizable as depth is efficiently computable. To ensure we correctly account just binary gates for the depth, we first assign the outgoing edges of each binary gate and negation gate with weight -1 and 0 respectively. Then, we simply take the circuit's underlying DAG, reverse its edges, and compute the shortest path from the output node to every other node in linear time [7].

We will sometimes work closely with specific parts of circuits. To this end we formally define a subcircuit rooted at a vertex α.

▶ **Definition 7** (Induced Subcircuit rooted at α). *Let $C = (V, E)$ be a circuit and let $\alpha \in V$. Let $P_\alpha = \{b \in V | \exists$ a path from β to $\alpha\}$. The induced subcircuit rooted at α in C, denoted $C[\alpha]$, is the vertex-induced subgraph of C whose vertex set is P_α and whose edge set consists of the edges in E_C whose endpoints are nodes in P_α.*

Table 3 Gate elimination rules, categorized by their impact on fanout

Fixing	Passing	Resolving	Pruning
$0 \wedge \gamma \to 0$	$1 \wedge \gamma \to \gamma$	$\gamma \wedge \neg\gamma \to 0$	$\gamma \wedge \gamma \to \gamma$
$\gamma \wedge 0 \to 0$	$\gamma \wedge 1 \to \gamma$	$\neg\gamma \wedge \gamma \to 0$	
$1 \vee \gamma \to 1$	$0 \vee \gamma \to \gamma$	$\gamma \vee \neg\gamma \to 1$	$\gamma \vee \gamma \to \gamma$
$\gamma \vee 1 \to 1$	$\gamma \vee 0 \to \gamma$	$\neg\gamma \vee \gamma \to 1$	
$\neg 0 \to 1$			$\neg\neg\gamma \to \neg$
$\neg 1 \to 0$			

B Gate Elimination

Most of the proofs in this paper involve arguments by gate elimination: we take a circuit, substitute a subset of inputs with constants, and then eliminate gates according to a set of basic simplification rules. This is illustrated in Figure 4.

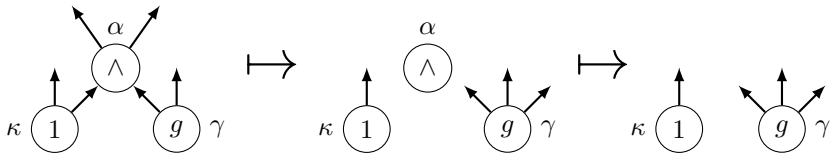


Figure 4 An example of applying the passing simplification $1 \wedge \gamma \to \gamma$. Notice that γ inherits the fanout from α, and α can then be garbage collected.

Towards algorithms for the f-Simple Extension Problem, we fix a concrete encoding of these circuit-manipulation steps. Gates γ are identified by natural numbers. We have one book-keeping manipulation: *garbage collection* deletes a gate γ with no outgoing wires; i.e, $\text{fanout}(\gamma) = 0$. A garbage collection step is encoded by $\langle \text{GC},\ \gamma \in \mathbb{N} \rangle$.

The main *gate elimination* rules are organized into four categories in Table 3, according to how they impact fanout. All these rules (i) eliminate at least one logic gate and (ii) have no effect on the function computed by a circuit C, because they are Boolean identities.

A gate elimination step is encoded by the tuple $\langle \text{GE},\ r \in 17,\ b : \mathbb{N} \to 7 \rangle$ where r identifies a rule and b is a "binding" that maps gates of C to gates of the circuit *pattern* on the left hand side of rule r. Each pattern is a well-formed Boolean formula and thus has a *main connective* α in the standard sense. To apply a rule, examine the right hand side: if it is

- **a constant** $c \in \{0, 1\}$**:** delete all incoming wires to α and change the type of α to c.
- **a matched node (i.e.,** γ**):** redirect all wires reading α to read from γ instead.

Note that the identifier of α **never** changes; only its type and incident wires. Depending on the initial fanout of each gate involved in the pattern, applying a rule could totally disconnect some gate(s) and leave the circuit in a state that needs garbage collection. We introduce notation for applying sequences of these steps to circuits, and define composed circuit manipulations that are "well behaved" with respect to specific circuits.

▶ **Definition 8.** *A* simplification *is a sequence of gate elimination and garbage collection steps. If λ is a simplification and C is a circuit, write $\lambda(C)$ to denote the new circuit obtained by applying each step of λ to C in order. If a single step of λ fails to apply in C, then fix $\lambda(C) = C$ so that invalid λ for C are idempotent by convention. A simplification λ is called*

Table 4 Changes to fanout after each type of gate elimination step.

Fixing	Passing	Resolving	Pruning
$\mathsf{fo}'(\kappa) = \mathsf{fo}(\kappa) - 1$	$\mathsf{fo}'(\kappa) = \mathsf{fo}(\kappa) - 1$	$\mathsf{fo}'(\alpha') = \mathsf{fo}(\alpha)$	$\mathsf{fo}'(\gamma) \geq \mathsf{fo}(\gamma) + \mathsf{fo}(\alpha) - 2$
$\mathsf{fo}'(\gamma) = \mathsf{fo}(\gamma) - 1$	$\mathsf{fo}'(\gamma) = \mathsf{fo}(\gamma) + \mathsf{fo}(\alpha) - 1$	$\mathsf{fo}'(\gamma) = \mathsf{fo}(\gamma) - 1$	$\mathsf{fo}'(\alpha) = 0$
$\mathsf{fo}'(\alpha) = \mathsf{fo}(\alpha)$	$\mathsf{fo}'(\alpha) = 0$	$\mathsf{fo}'(\neg) = \mathsf{fo}(\neg) - 1$	$\mathsf{fo}'(\neg) = \mathsf{fo}(\neg) - 1$

- terminal *for C if, for every λ' that extends λ by one additional gate elimination or garbage collection step, $\lambda(C) = \lambda'(C)$, and/or*
- layered *for C if the depth of **binary** gates binding to α in each step is non-decreasing.*

Simplifications formalize every sequence of circuit-manipulation steps except for substitution. They change the function computed by a circuit if and only if some input gate is garbage-collected. Simplifications suffice to impose convenient structural properties on arbitrary circuits.

▶ **Definition 9.** *A circuit C is* normalized *or* in normal form *if*

1. *C is constant-free **or** C is a single constant,*
2. *every gate in C has a path to the output, and*
3. *no sub-circuit of C matches the left hand side of a gate elimination rule.*

Any terminal simplification suffices to put a circuit C in normal form, and it straightforward to see that the number of constants in C lower-bounds the number of eliminated gates. We require more: every circuit C can be normalized by a *layered* simplification, and the number of eliminated gates is lower-bounded by the *cumulative fanout* of constants in C.

▶ **Lemma 10.** *For any well-formed circuit C with q constants that have total fanout ℓ, there is a terminal and layered simplification λ such that $\lambda(C)$ is a single constant **or** $|\lambda(C)| \leq |C| - \ell$.*

Proof. We update two potential functions as C is simplified: ϕ_t, the cumulative fanout of constants in C and μ_t, the number of binary gates eliminated after t manipulations. Since a circuit is normalized only when it is a single constant or $\phi_t = 0$, it suffices to exhibit a terminal layered simplification λ such that $\mu_t \geq \ell$ if $\phi_t = 0$.

Sort the gates of C by depth (breaking ties arbitrarily) to fix a total order on gates. This ordering remains consistent throughout the entirety of simplification. Constructing λ is straightforward: apply gate elimination rules in depth-order (by α) from the input gates of the circuit "up", garbage collecting any disconnected gate(s) immediately afterwards so only gates with a path to the output remain.

To update ϕ_t, we enumerate how fanout changes for the possible gates that may appear in each rule: the main connective α, a matched node γ, a constant κ, and a negation gate $\neg$. Write $\mathsf{fo}(\cdot)$ for the fanout of a gate before rule application, and $\mathsf{fo}'(\cdot)$ for the fanout of a gate afterwards. For example, in Figure 4, we have $\mathsf{fo}'(\alpha') = 0$, $\mathsf{fo}'(\kappa) = \mathsf{fo}(\kappa) - 1$, and $\mathsf{fo}'(\gamma) = \mathsf{fo}(\gamma) + \mathsf{fo}(\alpha) - 1$. We display the fanout-updates for each rule type in Table 4.

Because α has a path to the output, $\mathsf{fo}(\alpha) \geq 1$. Thus, each rule application reduces ϕ_t by at most 1, even when γ matches a constant (as in Figure 5). Any reduction in ϕ_t by 1 is accompanied by a corresponding increase in μ_t. Furthermore, observe that any subsequent garbage collection steps only increase μ and never reduce the ϕ. Any β that is garbage collected cannot read a constant: β would be the main connective for a rule application lower in the circuit than α. Hence if $\phi_t = 0$, we must have $\mu_t \geq \ell$.

Lastly, always choosing the depth-maximal α guarantees that our simplification is layered. This follows from the fact that rule applications only introduce new opportunities for gate elimination whose main connectives are shallower in the circuit. ◀

To finish capturing circuit manipulation, we encode a variable substitution step by the tuple $\langle \textsc{Sub},\ v \in \{x, y\},\ i \in \mathbb{N},\ c \in \{0, 1\}\rangle$ where v identifies the target set of variables (base x's or extension y's), i gives the variable number, and c is the value to be substituted. Apply a single substitution $\langle \textsc{Sub},\ v,\ i,\ c\rangle$ and to the circuit C by changing the type of every v_i-gate γ to the given constant c while **leaving the identifier of each γ intact,** exactly as in the application of a fixing rule.

Substitution *always* changes the function computed by C, restricting its domain to a subcube. Additionally, C remains well-formed after any substitution, because input gates have fanin 0. However, C is *never* in normal form immediately after a substitution because it introduces a constant. So we generalize simplifications by allowing for substitution steps.

▶ **Definition 11.** *A* restriction *is a sequence of substitution, gate elimination, and garbage collection steps. Restrictions generalize simplifications, so we extend notation and conventions accordingly: $\rho(C)$ denotes the result of applying restriction ρ to circuit C, and invalid restrictions for C are idempotent by convention.*

The categories of *terminal* and *layered* simplifications apply immediately to restrictions, because they explicitly reference only gate elimination or garbage collection steps. Thus, restrictions are terminal and/or layered only with respect to how they simplify and **not** how they substitute. When applying a restriction ρ to a Boolean function instead of a circuit, we simply ignore the additional simplification steps (if any).

C Structural Properties of Optimal Simple Extension Circuits

In this section we characterize the structure of optimal simple extension circuits. The main result is a *Y-trees decomposition* (Theorem 19), summarized below.

▶ **Theorem** (Informal). *In any optimal circuit for a simple extension, extension variables occur in isolated read-once formulas* (Y-trees) *that depend only on extension variables.*

▶ Remark 12. From this point onward, we exclusively consider the DeMorgan basis $\mathcal{D}$. The same arguments show an identical decomposition theorem in $\mathcal{R}$. In fact, $\mathcal{R}$ is simpler. Negations cannot be involved in any of our restrictions with keys, since Lemma 10 removes at least m *binary gates* from the circuit and no simplification steps introduce negations. Therefore, the removal of any negations would surpass the budgeted additional m gates in our complexity measure. Consequently, each Y-tree in an optimal $\mathcal{R}$ simple extension circuit is not only a read-once formula, it is also monotone.

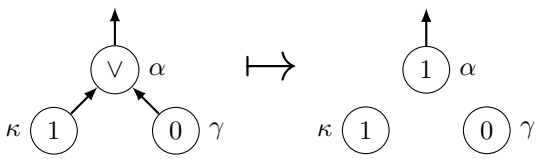


Figure 5 An application of a pruning rule where γ itself is a constant. Since $\mathsf{fo}(\alpha) \geq 1$, we have $\phi_{t+1} = \phi_t + \mathsf{fo}'(\alpha) - 2 = \phi_t + \mathsf{fo}(\alpha) - 2 \geq \phi_t + 1 - 2 = \phi_t - 1$.

C.1 Restrictions With "Speed Limits" on Gate Elimination

First we construct restrictions that are guaranteed to eliminate the gates of an optimal simple extension circuit as "slowly" as possible: exactly one costly logic gate is eliminated for each substituted variable. Structural information is then obtained by "watching" these slow restrictions operate. We begin by establishing basic properties of optimal simple extension circuits. First, combining normalization with the requirement that simple extensions are non-degenerate functions yields

▶ **Corollary 13** (Extension Variables are Read-Once). *In any optimal circuit for a simple extension, the fanout of each extension variable is exactly one. Furthermore, if an extension is negated, this negation has fanout exactly one.*

If any extension variable has fanout greater than one, substituting a key and normalizing (Lemma 10) would eliminate more than m gates. This leads to a contradiction: the resulting circuit computing f would have size less than $CC(f)$. Generalizing this observation, we obtain a short list of feasibly-checkable properties that guarantee a given circuit is the optimal circuit of a simple extension in Lemma 14.

Essentially, the existence of a circuit G computing g with size $CC(f) + m$ that (1) is not "trivially sub-optimal" and (2) not "trivially degenerate" witnesses that g is a Simple Extension of f. This is not immediate; such a circuit does not necessarily *guarantee* non-degeneracy of g and that $CC(g) = CC(f) + m$ as G could admit non-trivial simplifications. However, non-degeneracy of f implies that such straightforward witness circuits suffice.

▶ **Lemma 14.** *Let $f \in \mathcal{F}_n$ be non-degenerate and suppose $g \in \mathcal{F}_{n+m}$ is an* arbitrary *extension of f. If there exists a normalized circuit G computing g such that*

1. *$|G| = CC(f) + m$,*
2. *G reads each base variable at least once, and*
3. *G reads each extension variable exactly once,*

then g is a simple *extension of f and G is an optimal circuit for g.*

Proof. Let functions f, g and circuit G be as specified in the lemma. If g is non-degenerate in all variables, then G *must* be an optimal circuit for g — otherwise we could restrict G with any key to f in g and contradict the circuit complexity of f. So in this case g is indeed a simple extension of f, and G is optimal. Suppose now (towards contradiction) that g is degenerate with respect to some set of extension variables D.

We proceed by reverse induction starting with base case $|D| = 1$. Let y_i be the unique degenerate variable. By definition of degeneracy, $g|_{y_i=0} = g|_{y_i=1}$. Thus, there are keys to f in g with both $y_i = 1$ and $y_i = 0$. Therefore, we can substitute y_i to eliminate ≥ 1 costly logic gate α of G with a *fixing rule.* The matched gate γ loses a wire to α, so it may become disconnected. Run garbage collection recursively starting at γ, until no more fanout-0 gates exist in G, and call the resulting circuit G'.

Because y_i is the *only* degenerate variable, no other input gates can be disconnected by garbage collection after substituting into y_i (though some logic gates may be eliminated). So G' is a well-formed circuit with constant fanout ≥ 1 and size $|G'| \leq |G| - 1$. Furthermore, G' computes function g', an $(n + m - 1)$-variable extension of f, because the keys to f in g matching one setting of y_i are preserved in g'. Now substitute the remaining $(m - 1)$ extension variables of G' according to one of these keys to f in g' and normalize (Lemma 10) to obtain a circuit F^* that either

1. has at most $|G'| - (m - 1) + 1 < CC(f)$ gates, or
2. computes a constant function.

Both cases yield a contradiction; either to the circuit complexity or non-degeneracy of f.

Now suppose $|D| \geq 1$. Because all y-variables in D are degenerate, there are keys for all possible settings of them. So we reduce to the base case by substituting $|D| - 1$ extension variables such that, for each variable, exactly one costly logic gate is eliminated by a *passing rule.* Passing rules never disconnect the matched γ. This leaves a single degenerate y-variable in the new function g'', an $(n + m - |D| + 1)$-variable extension of f, computed by circuit G'' obtained by normalization. G'' and g'' are now exactly as specified in the statement of this Lemma, except for the guarantee that g'' has a unique degenerate extension variable. Therefore the base case with $|D| = 1$ applies to G'' and g'', concluding this proof. ◀

We will want to substitute key bits one by one to better analyze intermediate circuits. This analysis lends itself to induction only if the intermediate circuits themselves are optimal circuits for intermediate simple extensions. This motivates the following special class of restrictions.

▶ **Definition 15** (All-Stops Restrictions). *Let C denote an arbitrary circuit with $n + m$ input variables. A restriction ρ that substitutes m variables is* all-stops *for C if, for each $i \in m$, there is a prefix ρ_i of ρ such that $\rho_i(C)$ is an optimal circuit for a simple extension of the function computed by $\rho(C)$ on $(n + m - i)$ variables.*

If a restriction is all-stops, then exactly one binary gate is eliminated per substitution. Since each extension variable (or its negation) is read by a single binary gate, it is straightforward to analyze the simplifications that occur.

▷ Claim 16 (Simple Simplifications). Simplifications between substitutions in all-stops restrictions are *simple*, i.e. after substitution, the following gate elimination rules are applied in this exact order: a constant negation (if necessary), a passing rule, and lastly a double negation (if necessary).

The sole binary gate cannot be eliminated via a fixing rule, as otherwise the circuit is not constant free and must either become constant or we could eliminate more costly gates. These highly-structured manipulations will help us build up a robust structure for optimal simple extension circuits in the next subsection. However, we must first show that such restrictions even exist. We surpass this goal: we construct one that is also layered.

▶ **Lemma 17.** *Let $f \in \mathcal{F}_n$ and suppose $g \in \mathcal{F}_{n+m}$ is a simple extension of f. For any optimal circuit G computing g, there is an all-stops restriction ρ for G such that*

1. *ρ is terminal and layered, and*
2. *$\rho(G)$ is an optimal circuit computing f.*

Proof. Let Boolean functions f, g and circuit G be as in the Lemma. Sort the binary gates of G by depth (breaking ties arbitrarily) to fix a total order on gates. This ordering remains consistent throughout the entirety of our proof. Denote by $K \subseteq \{0,1\}^m$ the set of all keys to f in g. Normalize G to eliminate double negations — no other rules can apply because G is optimal (Lemma 10). Record these initial steps in ρ_0, which we then iteratively extend to the desired all-stops restriction.

Take y_i, read by the maximal in G. Such y_i are well-defined because G reads all extension variables exactly once (Lemma 14) and so each y_i has a unique depth in G. Consider the subcircuit around y_i: denote by α the **binary** main connective that reads $(\neg)y_i$ and by γ the sibling of $(\neg)y_i$ (the "matched gate" in elimination rules). Now let $K' \subseteq K$ be the subset of keys such that substituting y_i according to any $k \in K'$ would eliminate α with a passing rule.

If K' is non-empty, then substitute y_i according to any key in K' and normalize the resulting circuit. Record these steps in a restriction ρ_1 that extends ρ_0. The resulting circuit $G' = \rho_1(G)$ is in normal form, reads each base variable exactly as many times as G did, and reads each remaining extension variable exactly once. Therefore, the function computed by G' is a simple extension of f on $(n + m - 1)$ variables, and G' is an optimal circuit (Lemma 14). By construction, ρ_1 is layered and terminal, so we will continue extending ρ_1 by selecting a depth-maximal extension variable in G' and re-starting this case analysis.

If K' is empty, consider cases depending on the type of γ. Suppose towards contradiction that γ is not an extension variable or the negation of an extension variable; in this case there is no y_j with a path to γ because we selected a y_i of maximal depth. Substitute y_i according to any key in K, and observe that we obtain a circuit with total constant-fanout exactly $\mathsf{fo}(\alpha) \geq 1$ after applying a fixing rule. Then substitute all other extension variables according to the same key: the resulting circuit G'' has total constant-fanout $\mathsf{fo}(\alpha) + m - 1$. Normalizing, we contradict either the circuit size or non-degeneracy of f (exactly as in the proof of Lemma 16).

Therefore, if K' is empty, we must have that γ is $(\neg)y_j$ for some $j \neq i$ — a *layer tie* between extension variables. Thus, a restriction will be layered regardless of which substitution (into y_i or y_j) is made first. Using the particular structure of this sub-circuit (i.e., " $(\neg)y_j$ α $(\neg)y_i$ " where α is the main binary connective) we construct a key that sets y_j to eliminate α with a passing rule while preserving the rest of K. The idea is to extract information from G after y_i is substituted according to any key in K. Such substitution mutates the sub-circuit around y_i in G. If we can *identically mutate* G after setting y_j to eliminate α by passing *first*, then we are done. The details of this construction follow.

Purely to inspect the results, substitute y_i in G according to any key in K, and observe that we obtain a circuit G' with total constant fanout exactly $\mathsf{fo}(\alpha)$ after applying a fixing rule. Furthermore, the function computed by G' is an $(m-1)$-input extension of f where α has been substituted with a particular constant value $c_i \in \{0, 1\}$ and the variable y_j is disconnected from the input (and thus degenerate). This constant c_i is all the information we need.

Returning to G, there *must* exist a substitution $s_j \in \{0, 1\}$ for y_j that eliminates α via a passing rule: this follows immediately by inspecting the truth tables of binary $\{\wedge, \vee\}$ and case analysis on the occurrence of a negation gate between y_j and α. Extend ρ by first substituting s_j for y_j in G and normalize. The resulting circuit G'' passes all wires reading α to $(\neg)y_i$ and loses exactly one costly gate. Again by inspection, we can substitute y_i with some value $s_i \in \{0, 1\}$ such that the wires reading $(\neg)y_i$ read the constant c_i — identical to the subcircuit in G'. Thus, G' computes an $(m-1)$-input extension g_j of f witnessed by the key $\{y_j \mapsto s_j\}$. Extend the working restriction ρ_1 by first $\{y_j \mapsto s_j\}$ and then $\{y_i \mapsto s_i\}$, recording normalization steps in between and afterwards.

Finally, observe that G' meets all the preconditions of Lemma 14 and so g_j is a simple extension of f and G' is an optimal circuit computing g_j. The proof concludes by iterating this case analysis on the set of keys and depth-maximal extension variables until all extension variables are eliminated. ◀

C.2 Y-tree Decomposition

Equipped with layered all-stops restrictions, we can now inductively prove that optimal simple extension circuits have the following nice structure: each extension variable occurs in an isolated read-once subformula called a Y-tree that depends only on other extension variables, and the output of these read-once formulas is read by a single costly gate called a

combiner. Formally,

▶ **Definition 18** (Y-Tree Decomposition)**.** *Let G be a circuit with two distinguished sets of inputs:* base variables X *and* extension variables Y*. We say a triple $\langle \delta, b, T \rangle$, where δ—referred to as a* combiner*—is a binary gate in G, bit $b \in \{0, 1\}$ designates an input of δ, and T is a sub-circuit of G rooted at the b child of δ, is* admissible *if the following local conditions are met:*

1. *Each T is a read-once formula in only extension variables Y.*
2. *Each T is* isolated *in G — gate γ is the unique gate reading from T, and it only reads the root of T.*
3. *The sub-circuit of G rooted at the $\neg b$ child of γ contains at least one X variable.*

*A Y-*Tree Decomposition *of G is a set of admissible triples which are* non-intersecting*, that is, each $y_i \in Y$ appears in at most one T. The* size *of a decomposition is the number of tuples present in the decomposition. The* weight *of a Y-tree decomposition is the number of extension variables that are read in some T. We say a Y-tree decomposition is* total *if its weight is $|Y|$, i.e. every extension variable appears.*

We first remark that the binary gates spread across the read-once formulas and combiners in a total decomposition account for all m binary gates eliminated when substituting and simplifying with a key.

▶ **Theorem 19.** *If G is a minimal circuit for a simple extension, then G has a total Y-tree decomposition.*

Proof. We proceed via induction over m, the number of extension variables. When $m = 0$, the statement is vacuously true: the empty Y-tree decomposition is total for any optimal circuit of the base function.

Assume the statement holds for any simple extension with $k - 1$ extra variables. Let G_k be an optimal circuit for g_k, some simple extension of a function f, with k extension variables. By Lemma 17, there exists an all-stops layered restriction ρ such that $\rho(G_k)$ is optimal for f. By definition, there exists a prefix ρ_1 of ρ such that $G_{k-1} = \rho_1(G_k)$ is an optimal circuit for a simple extension of f with $k - 1$ extension variables. Without loss of generality, assume y_k is the variable substituted and eliminated by ρ_1. By our inductive hypothesis, G_{k-1} admits a total Y-tree decomposition, $\mathcal{Y}_{k-1}$. We will show how to add to/modify $\mathcal{Y}_{k-1}$ to obtain a total Y-tree decomposition for G_k by carefully considering the steps of ρ_1, .

By Corollary 16, ρ_1 is exactly the following sequence: a single substitution, (possibly) a constant negation, a passing rule, and (possibly) a double negation elimination. Consider this passing rule, and denote by α its the main connective, κ its constant, and γ its matched node. Observe that $(\neg)y_k$ is κ since, after substitution (and possibly constant negation), it is the only constant in the circuit. Let $(\neg)\beta$ be the matched node γ, i.e. the other input to α. We remark that ρ_1 only impacts the local neighborhoods of nodes adjacent to α; any triples of $\mathcal{Y}_{k-1}$ not involving these nodes are still admissible in G_k. We modify $\mathcal{Y}_{k-1}$ in the following ways, depending on whether β is present in any triples.

Adding a Triple: If β is not present in any triple, we argue that the triple $\langle \alpha, b, T' = (\neg)y_i \rangle$, where b denotes which input of α is $(\neg)y_i$, can be added to $\mathcal{Y}_{k-1}$. It is easy to see this triple is admissible: since ρ is layered, β can only depend on X variables, else there are deeper nodes reading extension variables that must be eliminated first. However, we must check that all triples of $\mathcal{Y}_{k-1}$ remain admissible in G_k. The only nodes of $\mathcal{Y}_{k-1}$ possibly impacted from G_k to G_{k-1} are combiners, since only those can read $(\neg)\beta$ in G_{k-1}. However, these combiners still fulfill the third condition in G_k: the sub-circuit

rooted at $(\neg)\alpha$ still depends contains X variables since $(\neg)\alpha$ reads $(\neg)\beta$. Hence all of these triples are admissible, and are non-intersecting—forming a total Y decomposition for G_k.

Modifying a Triple: It is not hard to see that the non-intersecting requirement enforces that β appear in at most one triple. Furthermore, if β is present in some triple, then it must *be* an extension variable: if it is an interior binary gate of some tree T, then there are deeper binary gates than α that read extension variables which are eliminated later in the layered ρ. When β is present in some triple of $\mathcal{Y}_{k-1}$, we will need to modify its tree to include y_k and α

Let $t = \langle \delta_i, b_i, T_i \rangle$ be the triple from $\mathcal{Y}_{k-1}$ that includes β. Not only is t inadmissible in G_k, it is not even well-formed: wires reading nodes of T_i in G_{k-1} are actually reading $(\neg)\alpha$ in G_k. As the modifications to G_k by ρ_1 are local, every other triple in $\mathcal{Y}_{k-1}$ remains admissible in G_k. We simply need to incorporate $(\neg)\alpha$ and $(\neg)\gamma$ into the triple, ensuring the first two conditions are met. Observe that there is only one node reading $(\neg)\alpha$, as otherwise the fanout of $(\neg)\beta$ will have greater than one fanout in G_{k-1}, violating Corollary 13. We modify t in two ways, depending on which gate in the triple is reading it.

- **α disconnects δ_i from T_i:** If δ_j reads $(\neg)\alpha$, then observe it, not the root of T_i, is the b_i child of δ_j. However, the sub-circuit T' rooted at $(\neg)(\alpha)$ is a read-once formula in only extension variables. Replacing T_i with T' in t repairs the triple and produces a total decomposition.
- **α disconnects β inside T_i:** If a node in T_i reads $(\neg)\alpha$ in G_k, then T_i is not actually a well-formed sub-circuit of G_k. However, the sub-circuit T' rooted at the b_i child of δ_i is still read-once formula over extension variables, once we include $(\neg)\alpha$ and $(\neg)y_k$. Hence $\langle \gamma_j, b_j, T' \rangle$ can replace t to produce a total decomposition $\mathcal{Y}_k$.

◀

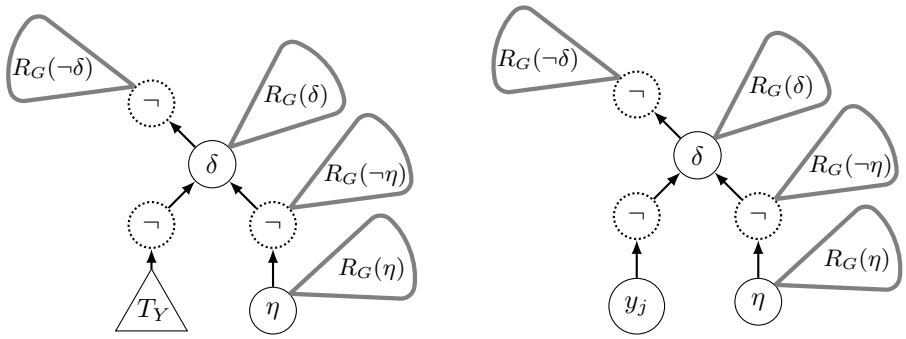
D The f-Simple Extension Problem is Fixed-Parameter Tractable

In this section we build a polynomial time algorithm for the XOR-Simple Extension Problem. By Lemma 14, it suffices to find a normalized circuit for g of size $CC(f) + m$ which reads every extension variable.

For exposition, recall the framing of encoding simple circuit extensions from Section 1.4.2: suppose g is a simple extension of f and Alice knows G, an optimal circuit for g. Alice can obtain an optimal circuit F computing f by simply restricting the y-variables of G with a key and performing gate elimination. Now consider the following communication problem: Bob knows F, and Alice would like to send him G using as few bits as possible. Because g is a simple extension of f, Alice can compute the Y-tree decomposition of optimal circuit G. The idea is to send Bob a sequence of instructions that tell him exactly how to graft each Y-Tree of G onto the gates of F.

D.1 Grafting Y-Trees: Key Ideas & Structural Properties

We begin by illustrating the "grafting" idea for the algorithm. First consider the case where there is only a single Y-tree $T_{\mathcal{Y}}$ in the decomposition of G. Then there must be some costly gate η — an *origin* already present in F — that is combined with $T_{\mathcal{Y}}$ in G. Our decomposition theorem shows that every possible arrangement of this *graft* is depicted in



(a) Before any extension variable restrictions **(b)** All extension variables restricted except y_j

Figure 6 The local neighborhood of a combiner δ and Y-tree $T_{\mathcal{Y}}$ grafted on an origin η.

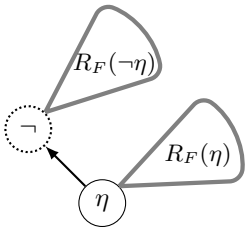


Figure 7 The origin η after the Y-tree has been pruned.

Figure 6, which shows the local neighborhood[7] of the single Y-tree in G.

In Figure 6, both δ and η represent costly gates that *must* be present, the dotted negations represent NOT gates that *may* be present, and the cones represent connections to the rest of G. The notation $R_G(\beta)$ means "the set of costly gates that read gate β in circuit G." These sets are depicted by shaded cones, because at least one of them must be non-empty — otherwise, G would not be an optimal circuit — but we do not know which. Note how the Y-tree and potential negation atop it are *isolated* from the rest of the circuit, except for connections via the *combiner* gate δ. We will describe in some detail how this single graft can be transmitted and sketch the issues involved in efficiently coding *all* grafts and Y-trees for Bob.

First, Alice applies gate elimination to G using a prefix of a terminal and layered all-stops restriction (Lemma 17), eliminating all but one of the y-variables in $T_{\mathcal{Y}}$. Because $T_{\mathcal{Y}}$ is an isolated formula in G, a single variable y_j is left in G at the end of this process: the local neighborhood transforms from Figure 6a to Figure 6b. Crucially, all gates outside $T_{\mathcal{Y}}$ are unaffected. Now, Alice runs a final step of gate elimination, substituting y_j according to the key. The result will be as depicted in Figure 7: the local neighborhood of η in F. The key observation is that *Bob has perfect knowledge of this structure* — it is simply a sub-circuit of F. If Alice can send (1) an identifier for η (2) a "diff" between the neighborhood of η in F compared to G (Fig. 6b vs. 7) and (3) a description of $T_{\mathcal{Y}}$ this would suffice for Bob to efficiently transform F into G.

The most straightforward protocol would send $O(\log(\#\mathsf{gates}(F)))$ bits to identify η for Bob. This is too expensive if there is more than one Y-tree in G. Recall that we can only

[7] Think of this as "zooming in" to inspect one of the shaded circles in Figure 2.

tolerate runtime of $2^{O(n+m)}$ and intend to brute-force all possible codes. There could be as many as m Y-trees with a single variable each and thus m origins, so brute-forcing this simple code would require time $2^{O(m\log(n)+n)}$ for $\#\mathsf{gates}(F) = O(n)$ — super-polynomial in $2^{(n+m)}$ and therefore unacceptable.

Instead, Alice can send a $\#\mathsf{gates}(F)$-bit origin indicator vector χ, where χ_i is 1 if gate i of F is the origin of some Y-tree. It is feasible to brute-force these vectors in $2^{O(n)}$-time for any possible number of origins given a linear upper bound on the circuit size of f. Returning to the case where G has a single Y-tree, Bob identifies η by reading the single 1-bit of χ. For the local neighborhood of η, the following information must be transmitted to summarize the "diff" between F and G:

1. The costly functions computed by gates η and δ,
2. presence or absence of each possible NOT gate depicted in Figure 6b,
3. whether η is connected to the left or right input of δ,
4. the description of $T_{\mathcal{Y}}$, and
5. the sets of gates that read from each individual element of the graft, $R_G(\cdot)$.

Items 1 - 3 can be described by a constant-length bitstring, depending only on the enumeration of all possible "graft" sub-circuits implied by our characterization of gate elimination. We code the exact Y-Tree $T_{\mathcal{Y}}$ (item 4) explicitly for now and eliminate it later. Here, we are concerned with how to efficiently code the sets $R_G(\cdot)$ without sending explicit "pointers" to nodes of F, which remain too expensive per the discussion of transmitting η above.

To code these wire movements efficiently, Alice will exploit the relationship between the gates reading from η in F and the gates reading from η in G as determined by gate elimination. Bob knows the contents of $R_F(\eta)$ and $R_F(\neg\eta)$ — the sets of costly gates reading from $(\neg)\eta$ in F. Collect the gates that read from the graft, in both G and F:

$$R_G = R_G(\eta) \cup R_G(\neg\eta) \cup R_G(\delta) \cup \mathsf{R}_G(\neg\delta)$$
$$R_F = R_F(\eta) \cup R_F(\neg\eta)$$

Observe that $R_G \subseteq R_F$ because during the single run of gate elimination that transforms Figure 6b into Figure 7, all the wires reading the eliminated gate δ are "inherited" by η. Furthermore, *Bob knows η from χ and thus can reconstruct R_F*, so Alice can identify any wire relevant to the graft by coding "the j-th element of R_F." Here we must apply the assumption that optimal circuits for f have at most constant fanout, independent of n, to bound the length of these *relative* wire-identifiers by a constant. Thus, Alice can identify *exactly which* wires should be moved from η to read $(\neg)\delta$ in G. Inspecting Figure 6b again, she can also code *where* they move using constantly many bits, because there are only two options: reading from $\neg\delta$ or δ directly.

This discussion has outlined the base case of our encoding/decoding argument, where only a single Y-tree is transmitted to Bob. Notice that, if there are multiple *combiner-disjoint* Y-trees in G, an essentially identical strategy could encode all these Y-trees. However, reverse gate elimination can create somewhat more complex structures: specifically, we can have combiners δ_i grafted onto *a distinct combiner* δ_j, instead of a gate η that was present in the original circuit F. These *compounded combiners* correspond exactly to the "adding a triple" case in the proof that Y-Tree decompositions exist (Theorem 19) where α occurs *between* β and an existing combiner.

Even so, Alice can use the fact that every combiner δ_i has a *unique* origin η to identify the "first" combiner grafted onto η and instruct Bob to add subsequent η-derived combiners in depth-respecting order. We now prove the required properties of origins and combiners.

▶ **Definition 20** (Original & Originating Gates). *Let G and F be circuits, ρ be an all-stops restriction, and suppose $\rho(G) = F$. We call a gate β of G* original *if it is not garbage collected by ρ — i.e., if $\beta \in F$. Furthermore, let G have a Y-tree decomposition with elements $\langle \delta,\ b,\ T \rangle$. The* origin *of each δ is the depth-maximal original gate η such that X variables occur in $G[\eta]$ and there is a path from η to the output of G through δ.*

The origin of a combiner can be depth-trivial: a single input gate x_i or the unique output gate are both valid origins. Intuitively, if η is the origin of δ, then δ "takes" wires from η when it is grafted into the circuit. When the maximum fanout of an optimal circuit is constant, this imposes a hard limit on the number of compounded combiners that share the same origin.

▶ **Lemma 21** (Every Combiner has a Unique Origin). *Suppose G is a circuit with Y-tree decomposition $\mathcal{D}$, and let $\langle \delta,\ b,\ T \rangle$ be an arbitrary element of $\mathcal{D}$. Then δ has exactly one origin in G.*

Proof. We argue by induction on the structure of G, exploiting that the DeMorgan basis has fan-in two. We will walk "down" the circuit from δ until an original gate is encountered. By definition, δ has only two inputs: input b leads to a Y-tree, and input $\neg b$ leads to a sub-circuit containing X variables. Thus we must go down the $\neg b$ argument to find the origin. Argument $\neg b$ = gate β is either another combiner in the decomposition or not — if β is not a combiner, then it is an original gate, and thus unique because the path to β from δ was fully determined. β is topologically maximal because it was the first non-combiner encountered walking "down" the circuit in topological order. Then, if argument $\neg b$ is another combiner, we are done by induction: there is simply another deterministic choice to find a sub-circuit where X variables occur, because argument b of β is also a Y-tree. ◀

Finally, observe that there are only constantly-many ways to eliminate a binary gate by a simple simplification (Claim 16) as depicted in Figure 8. This follows by brute-forcing all possible sequences of simple simplifications around eliminating one costly gate, formalizing the idea of a particular "graft" sub-circuit from earlier in this section: call each possible realization of sub-circuits a *widget*.

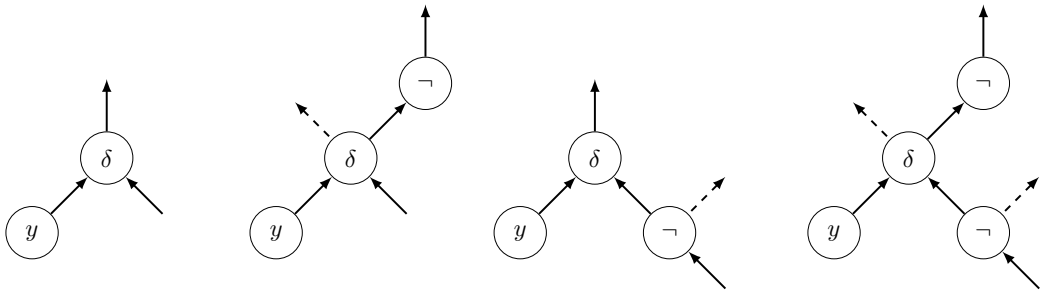


Figure 8 The four possible widgets in a splice code (up to symmetry). The node labeled y represents which child of the combiner δ is the Y-tree. A dotted edge not leading to another node represents one (or more) possible connections to a *costly* gate outside of the widget. A solid edge not leading to another node represents one (or more) guaranteed connections to a *costly* gate outside of the widget.

D.2 Grafting Y-Trees: Efficient and Explicit Encodings

Suppose η is a gate in some circuit C, and let the gates $\zeta_1, \ldots, \zeta_{\leq \ell}$ read η in C. We use *η-relative fanout-coding* to name another gate β in a circuit C' obtained by transforming C by writing an ℓ-bit vector v where:

$$v_i(\beta, \eta) = \begin{cases} 1, & \text{if } \zeta_i \text{ reads } \beta \text{ in } C' \\ 0, & \text{otherwise} \end{cases}$$

Trivially, when $C = C'$ and we fanout-code η relative to itself, the result is $1^{\mathsf{fo}(\eta)} \circ 0^{\ell - \mathsf{fo}(\eta)}$. Suitability for fanout-coding is the prime motivation of our requirement that gate identifiers are **never** mutated (only recycled or discarded) by circuit manipulation.

▶ **Definition 22** (Splice Codes). *Suppose F is a circuit with maximum fanout ℓ. A splice code E is a sequence of tuples that encode a transformation of F into G, a circuit for a simple extension of F. The splice code begins with an **Origin Indicator Vector:** a bitvector of length $CC(f)$ where entry i is set to 1 iff gate i of F is an origin. Then, for **each** origin η, a **sequence of** **splices** follows — one splice per combiner originating in η. A splice is:*

1. ***Target Gate:** Which gate should be grafted onto? Written as η-fanout-relative code.*
2. ***Selected Wires:** Which wires does the spliced widget take from the target gate? Written as ℓ-bit indicator vector, where 1-bits must be a subset of the target gate's code.*
3. ***Widget:** A constant number of bits naming one of constantly-many combiner widgets.*
4. ***Wire Moves:** Map from taken wires taken to gates of the widget: at most $(\ell - 1)$ constant-length identifiers.*
5. ***Explicit Y-Tree:** A fully specified read-once formula in the Y variables.*

▶ **Theorem 23** (Splice Codes for Simple Extensions). *Suppose $g \in \mathcal{F}_{n+m}$ is a simple extension of $f \in \mathcal{F}_n$ and let $\tilde{G}$ be an isomorphism class of minimal circuits for g. Then, there is an "unlocked" isomorphism class $\mathsf{UL}(\tilde{G})$ of minimal circuits computing f such that, for every representative F of $\mathsf{UL}(\tilde{G})$, there is a* splice code *E such that an efficient algorithm $\mathtt{Decode}(F, E)$ prints a circuit in the isomorphism class $\tilde{G}$. The length of E is $O(\ell \cdot |F| \cdot m \cdot \text{bits}(Y))$, where ℓ is the maximum fanout of F and $\text{bits}(Y)$ is the number of bits required to code a Y-tree on m variables.*

Proof. Take a representative G of the isomorphism class of ciurcits $\tilde{G}$ with gates named such that they are compatible with depth and a topological order. Let $\mathcal{D}$ be the Y-tree decomposition of G (Theorem 19) and let ρ be the particular all-stops restriction of G implicit in the proof of Theorem 19 such that $F = \rho(G)$ computes f and F is optimal for F. We will "reverse" ρ to obtain a circuit isomorphic to G from F.

First, list all the origins in G — it is immmediate from the definition and inspecting ρ that this is possible. Now, if the set of combiners associated with each origin is of size 1, it is clear that the trivial fanout-relative code is used to set the Target Gate for each combiner, and the legnth lower bounds are immediate because the overhead of encoding sequences is linear, and all fields of the sequences are linear in ℓ, a constant, or proportional to the size of a particular Y-tree. Origin gates are **never** named explicitly; the order of sequences of splices suffices to associate them with the appropriate splice-sequence.

Suppose now that some gates of G originate q combiners where $1 < q \leq m$ and fix an arbitrary such origin η. To order the splices, perform breadth-first-search starting from η in G and observe which wires of η have been "spliced" with which combiners in each stop of the all-stops restriction ρ. Using this ordering and the sub-sequences of ρ where combiners originating in η are eliminated, observe that at each step there is there is a set of

maximally-shallow *available combiners* that could be grafted to: those between η and the output of the circuit which feed into a gate that used to be fed by η directly.

Each available combiner must be uniquely identified by an η-relative fanout-code, because otherwise an intermediate normal circuit ρ would not be size-optimal: we could restrict using the key embedded in ρ to find that the same gate must read η twice, which triggers a resolving gate-elimination. Therefore, even compound-combiners can be uniquely regenerated by splice codes.

To conclude the proof, observe that the isomorphism class of F is exactly $\mathsf{UL}(\tilde{G})$, as desired, because all coding operations relied only on being given a circuit isomorphic to F whose gates were named in depth-sorted and thus topological order. This is because our formalization of circuit manipulation never introduces "fresh" identifiers; it only deletes or recycles them. ◀

D.3 The Final Ingredient: Truth-Table Isomorphism

There are two issues with trivial brute-force over splice codes. First, there are too many base circuits for XOR_n: there are at least as many optimal XOR_n circuits as there are labeled rooted binary trees with n leaves. Let C_n be the n^{th} Catalan number; there are $n!\,C_{n-1} = 2^{\omega(n \log n)}$ labeled rooted binary trees [42]. Similarly, there are $m!\,C_{m-1}$ explicit Y-trees reading all m variables. However, in both expressions, the dominating factorial terms $n!$ and $m!$ arise from permuting the labels of the variables. But two circuits which can be transformed into one another by permuting inputs compute *truth-table isomorphic functions.*

▶ **Definition 24** ([30, 3]). *Two functions $f, g \in \mathcal{F}_n$ are* truth-table isomorphic *if there exists a permutation π on $[n]$ such that $g(x) = f(\pi(x))$.*

It is straightforward to connect truth-table isomorphism with permuting circuit inputs.

▶ **Observation 25.** *The following two statements, the first a universal statement and the second existential, describe the relationship between permuting circuit variable labels and truth-table isomorphism.*

(1) *If a Boolean function $f \in \mathcal{F}_n$ is truth-table isomorphic to another Boolean function $g \in \mathcal{F}_n$ then* ***any circuit*** *for f can be transformed into some circuit for g by relabeling input x_i with $x_{\sigma(i)}$ for all $i \in [n]$ using some permutation σ on $[n]$.*

(2) *If* ***a given circuit*** *for $f \in \mathcal{F}_n$ can be transformed into* a *circuit for $g \in \mathcal{F}_n$ by relabeling input x_i with $x_{\sigma(i)}$ for all $i \in [n]$ using some permutation σ on $[n]$ then f is truth-table isomorphic to g.*

Proof. We first prove (1). Let π be the witnessing permutation between f and g, i.e. $f(x) = g(\pi(x))$. Observe that $f(\pi^{-1}(x)) = g(\pi(\pi^{-1}(x))) = g(x)$. Let F be any circuit for f. Let G be the circuit obtained by relabeling each input x_i by $x_{\pi(i)}$. We wish to argue G computes g. Observe that evaluating g on input x is the same as evaluating F on $\pi^{-1}(x)$ since $x_{\pi(\pi^{-1}(i))} = x_i$. Therefore $G(x) = F(\pi^{-1}(x)) = f(\pi^{-1}(x)) = g(x)$.

For (2), fix some optimal circuit F and let σ be the witnessing permutation used to relabel F to obtain G, a circuit for g. We have $f(x) = F(x) = G(\sigma^{-1}(x)) = g(\sigma^{-1}(x))$. As σ is a permutation on $[n]$, σ^{-1} is also a permutation on $[n]$ and thus f and g are truth-table isomorphic. ◀

The implication is that do not need to try every possible base XOR_n circuit or even try every explicit splice code. We can take an unlabeled base circuit and unlabeled Y-trees and assign variables arbitrarily. We then just need to check whether the function our reconstructed

circuit computes is isomorphic to g. This is feasible because (rather surprisingly) truth-table isomorphism testing can be done in polynomial time.

▶ **Theorem 26** (Corollary 1.3 of [30]). *Given the truth-tables for two Boolean functions, testing whether they are equivalent under permutation of variables can be done in time $c^{O(n)}$ where c is a constant.*

To this end we formally define "unlabeled" optimal circuits.

▶ **Definition 27** (Open Optimal Circuit). *A circuit is* open *if all of its inputs are unlabeled. An open circuit is optimal for a Boolean function f if there exists some labeling of its' inputs so that the resulting circuit is an optimal circuit for f.*

D.4 An Efficient Simple Extension Problem Solver for XOR

The following algorithm, when given L, a complete list of representatives from each open isomorphism class of optimal circuits for f, decides the f-Simple Extension Problem. By "implicit splice code," we mean a splice code without the Y-trees specified.

Algorithm 2 `Ckt-SE-Solver`$(n \in \mathbb{N},\ g \in \mathcal{F}_{n+m}, L)$

1: Verify $\exists \rho \in \{0,1\}^m$ such that $g|_\rho \equiv f$ and **return False** if not
2: Verify g is non-degenerate and **return False** if not
3: **for** each open circuit F in L **do**
4: $s \leftarrow$ number of costly gates in F
5: label the open nodes of F by an arbitrary permutation of $x_1, \ldots, x_n$
6: **for** each implicit splice code E of length at most $m + s$ **do**
7: $d \leftarrow$ number of combiners recorded in E
8: **for** each $a_1, \ldots a_d \in \mathbb{N}$ s.t. $\sum_{i=1}^{d} a_i = m$ **do** ▷ Valid distribution of variables to Y-trees
9: **for** each d-tuple of read-once formulas with $a_1, \ldots a_d$ open nodes **do**
10: label the open nodes of each read-once formula by an arbitrary permutation of y-inputs
11: insert the read-once formulas into combiner instructions of E in lexicographic order
12: $\tilde{G} \leftarrow$ `Decode`(F, E)
13: **if** `tt`$(\tilde{G}) \simeq$ `tt`(g) **then** ▷ Test using the procedure of Theorem 26
14: **return True**
15: **return False**

▶ **Theorem 28.** *Given L is a list with a representative from every optimal open circuit class of f, Algorithm 2 decides the f-Simple Extension Problem in time $|L| \cdot 2^{O(\ell(s+m))}$ where ℓ is the maximum fanout of any node in any circuit in L and $s = CC(f)$.*

Proof. We first prove completeness, i.e. that if g is a simple extension then Algorithm 2 returns true. By definition of simple extension, the checks in steps 1 and 2 pass. Since g is a simple extension, $CC(g) = s + m$; fix any optimal circuit G for g. By Theorem 23, there is an explicit splice code $\hat{E}$ and circuit isomorphism class representative $\hat{F}$ such that `Decode`$(\hat{F}, \hat{E})$ produces a circuit $\hat{G}$ isomorphic to G. Let $\hat{T}_1, \ldots \hat{T}_t$ be the explicit Y-trees in $\hat{E}$. During some iteration of the algorithm (1) the implicit splice code will be consistent with E and (2) F and $T_1, \ldots T_t$, the chosen open read-once formulas, will be isomorphic to

$\text{open}(\hat{F}), \text{open}(\hat{T}_1), \ldots \text{open}(\hat{T}_t)$ (where $\text{open}(C)$ is circuit C with its input labels stripped away). In this iteration, let F' and E' be the arbitrary explicit completion of the open circuit F and implicit splice code E chosen by the algorithm. Let $G' = \texttt{Decode}(F', E')$ and observe $\text{open}(G')$ is isomorphic to $\text{open}(\hat{G})$. Thus there is a permutation of the variables of G' such that the resulting circuit is isomorphic to $\hat{G}$. By part (2) of Observation 25, the function computed by G' is truth-table isomorphic to g and thus the algorithm accepts.

For soundness, observe that if the algorithm accepts then it passed step 1 and 2 which demonstrate the existence of a key and non-degeneracy of g. Finally in steps 12 and 13, the algorithm must have constructed a constant-free circuit of size $s + m$ which computes a function truth-table isomorphic to g. Applying part (1) of Observation 25 we can transform this circuit into a circuit for g by relabeling inputs. This constant-free circuit of size $s + m$ for g, in conjunction with the fact f is non-degenerate and the existence of a key, guarantees $CC(g) = s + m$ by Lemma 14. Therefore, by definition, g is a simple extension of f.

For the running time, we first observe that steps 1 and 2 take $2^{O(s+m)}$ since the algorithm just verifies whether one of the 2^m restrictions yields $tt(f)$ (and $s = \Omega(n)$) and verifies for each of the m extension variables there is an assignment where flipping the value of the variable changes the output of g. There are then $|L|$ iterations where each run in time $2^{O(\ell(s+m))}$ where ℓ is the maximum fanout of any node in any circuit in L as the algorithm builds this number of explicit splice codes. The algorithm tries every explicit splice code where y-inputs are labeled arbitrarily (say, in increasing order). The number of such codes is $\sum_{d=1}^{m} r(d)$ where $r(d)$ is the number of such splice codes with d combiners. We see

$$r(d) = \sum_{\substack{a_1+\ldots+a_d=m \\ a_i \in \mathbf{N}^+}} \prod_{j=1}^{d} t(a_j),$$

where $t(a_j)$ is the number of read-once formulas with a_j open nodes.

We first bound $t(a_j)$. A number of read-once formula with a_j open nodes corresponds to the number of rooted binary trees where each internal node is labeled $\wedge$ or $\vee$ and each edge is weighted 0 or 1 (representing if there is a negation between those two nodes). The number of unlabeled rooted binary trees with n leaves is C_{n-1}—the $(n-1)^{\text{th}}$ Catalan number [42]. Thus $t(a_j) = C_{a_j-1} \cdot 2^{a_j-1} \cdot 2^{2(a_j-1)+1} = C_{a_j-1} \cdot 2^{2a_j}$ where the latter factors are from labeling the internal nodes and edge weights (including the output edge) respectively. Since $C_{a_j-1} < 4^{a_j}$, we have $t(a_j) < 2^{5a_j}$ [42]. Since the a_j sum to m, then $\prod_{j=1}^{d} t(a_j) < 2^{5m} = 2^{O(m)}$. The number of solutions to $a_1 + \ldots + a_d = m$ where $a_j > 0$ is $\binom{m+d-1}{d-1}$ [8]. Notice that $\binom{m+d-1}{d-1} \leq \sum_{i=0}^{m+d-1} \binom{m+d-1}{i} = 2^{m+d-1}$ [8]. Since $k < m$ then this is $2^{O(m)}$. Therefore $r(d) < 2^{O(m)}$ and thus the number of explicit splice codes is at most $m2^{O(m)} = 2^{O(m)}$.

Once an explicit splice code is constructed, constructing $\tilde{G}$ with the decoder takes $\text{poly}(s+m)$ time and testing truth table isomorphism takes $O((s+m)^2) \cdot 2^{O(n+m)} \cdot 2^{O(n+m)} = 2^{O(n+m)}$ as the algorithm has to write the truth-table and then run the algorithm from Theorem 26. Since $s = \Omega(n)$ this takes $2^{O(s+m)}$. Overall the running time is therefore $|L| \cdot 2^O(\ell \cdot (s + m))$. ◀

In general, $|L|$ may be exponential in $|tt(g)|$, ℓ may be $\omega(1)$, and s may be superlinear in n. As such Algorithm 2 is not a polynomial time algorithm for the Simple Extension problem in general. However, as noted in Corollary 33, these parameters for XOR_n are tractable. This yields our main theorem:

▶ **Corollary 29.** *The Simple Extension Problem with the base function $f = \mathsf{XOR}_n$ is in* P

E Optimal XOR Circuits Are Binary Trees of XOR_2 Sub-Circuits

Algorithm 2 of Section D can only rule out a particular f from showing hardness for MCSP via f-SEP when the optimal circuits for f are well-understood. It requires an exact characterization of the set of all optimal circuits—a requirement that currently is rarely fulfilled.

One of the earliest studied explicit Boolean functions, XOR, is one of the only functions for which we even have partial fulfillment of this ornery prerequisite. Recall the definition XOR, the parity function.

▶ **Definition 30** (XOR). *For* $x \in \{0,1\}^n$, $\mathsf{XOR}_n(x) = \begin{cases} 1 & \text{if an odd number bits of } x \text{ are } 1 \\ 0 & \text{otherwise.} \end{cases}$

Schnorr proved one of the first explicit circuit lower-bounds on XOR.

▶ **Theorem 31** ([39]). XOR_n *requires at least* $3(n-1)$ *gates in the DeMorgan basis.*

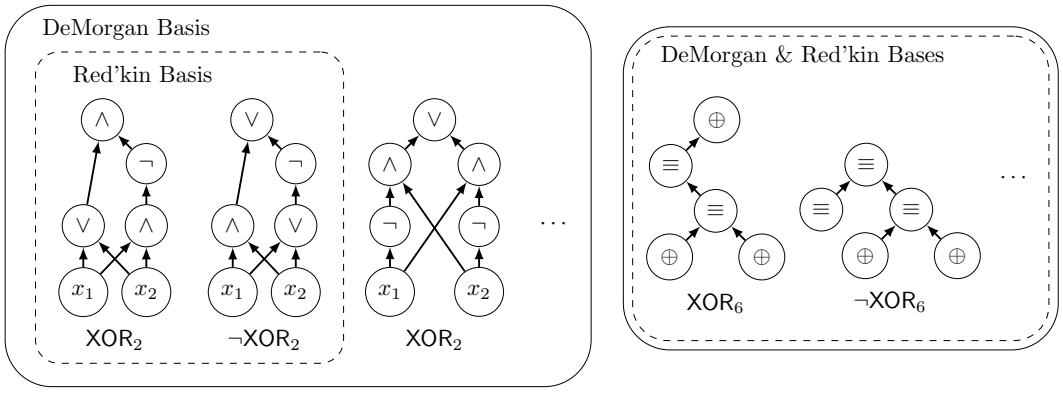
This lower-bound in fact has a matching upper bound and the construction is straightforward: take any binary tree with n leaves labeled $x_1, \ldots, x_n$ where the $n-1$ interior nodes have been labeled by $\oplus$ and replace the $\oplus$ nodes with any circuit of size 3 that computes XOR_2. Furthermore, since $\neg$ gates do not count towards the circuit size, any circuit for XOR_n can easily be transformed into an equal size circuit computing $\neg\mathsf{XOR}_n$ and vice versa. Combining these observations yields:

▶ **Corollary 32** ([39, 43]). *A circuit* C *computing* $(\neg)\mathsf{XOR}_n$ *is optimal if and only if* $|C| = 3(n-1)$.

However, this leaves open *how* those $3(n-1)$ gates can be arranged. Are there optimal circuits for XOR that do not follow the upper-bound construction? Previous work showed that in $\mathcal{R}$, all optimal circuits follow this construction: optimal XOR_n circuits are binary trees of $(n-1)$ XOR_2 sub-circuits [25]. In this section, we extend this characterization, as seen in Figure 1, to $\mathcal{D}$.

▶ **Theorem** (Informal Statement of Theorem 37). *Optimal* $(\neg)\mathsf{XOR}_n$ *circuits in* $\mathcal{D}$ *partition into trees of* $(n-1)$ $(\neg)\mathsf{XOR}_2$ *sub-circuits.*

This structural characterization implies the crucial combinatorial parameter bounds that are required to apply Algorithm 2 and therefore rule out the possibility of proving hardness of MCSP via XOR-SEP.



▶ **Corollary 33.** *The following properties hold for* XOR_n*, where* $n \geq 1$

- *The size of optimal circuits computing* XOR_n *is linear (Corollary 32).*
- *The maximum fan-out of such circuits is a constant.*
- *The number of optimal circuits for* XOR_n*, up to permutation of variables, is* $2^{O(n)}$*.*

Before we prove our main result of this section, we introduce some auxiliary facts that we will repeatedly require.

E.1 Basic Properties of XOR

We first give two basic facts about $(\neg)\mathsf{XOR}_n$ that are immediate consequences of the definition.

▶ **Fact 1** ($(\neg)$XOR is Fully DSR)**.** XOR_n *is* fully downward self-reducible*, i.e. for any input* $x \in \{0,1\}^n$*, any non-empty sets* S *and* T *partitioning* $[n]$*,*

$$\mathsf{XOR}_n(x) = \mathsf{XOR}_2(\mathsf{XOR}_{|S|}(x_S), \mathsf{XOR}_{|T|}(x_T))$$

where $x_S = \{x_i : i \in S\}$ *and* $x_T = \{x_i : i \in T\}$*. Furthermore, this means for any assignment* α_S *of variables in* x_S*,* $\mathsf{XOR}_n(x)|_{\alpha_S} = (\neg)\mathsf{XOR}_{|T|}(x_T)$*. The same is also true of* $\neg\mathsf{XOR}_n(x)$

▶ **Fact 2** (All Subfunctions of $(\neg)$XOR are Non-Degenerate)**.** $(\neg)\mathsf{XOR}_n$ *not only depends on all of its inputs but it is also* maximally sensitive*, i.e. for all* $i \in [n]$*, for inputs* $x \in \{0,1\}^n$*,* $(\neg)\mathsf{XOR}_n(x) \neq (\neg)\mathsf{XOR}_n(x \oplus e_i)$*.*

Combining Facts 1 and 2, if we substitute for a single variable in a $(\neg)\mathsf{XOR}_n$ circuit where $n \geq 2$ are guaranteed to get a circuit which computes $(\neg)\mathsf{XOR}_{n-1}$. Applying the tight version of Schnorr (Corollary 32) we see that subsequent simplification cannot remove more than three costly gates. Formally,

▶ **Corollary 34** (Elimination Rate Limit for Optimal XOR Circuits)**.** *Let* C *be an optimal circuit computing* $(\neg)\mathsf{XOR}_n$ *where* $n \geq 2$*. Let* C' *be the circuit after substituting* $x_i = \alpha$ *for some* $i \in [n]$ *and* $\alpha = \{0,1\}$ *and applying simplifying. We have that* $|C'| \geq |C| - 3$ *and* C' *is not constant.*

We will repeatedly apply this corollary in our proof: deviation from the prescribed structure will often allow us to substitute and remove more than three distinct binary gates. The other main source of contradictions will be substitutions and rewrites that disconnect inputs (violating Fact 2) or that leave inputs with exactly one costly successor. This violates the fact that XOR_n reads each of its inputs twice.

▶ **Lemma 35** ($(\neg)$XOR is Read-Twice (Folklore))**.** *Let* C *be a normalized optimal circuit computing* $(\neg)\mathsf{XOR}_n$ *where* $n \geq 2$*. The fanout of every variable* C *is exactly 2.*

▶ Remark 36. In the proofs of this section, we will omit negations whenever reasonable, since we are more interested in the binary gates which solely contribute to circuit size in $\mathcal{D}$. We write α is a *costly successor* to β to mean $(\neg)\beta$ is an input to binary gate α.

Proof. Let C be an optimal normalized circuit computing $(\neg)\mathsf{XOR}_n$ for arbitrary n. Suppose there is a variable x_i whose fanout is not 2. There are two cases: (1) the fanout of x_i is 1 and (2) the fanout of x_i is at least 3.

(1) Take α, the assumed unique costly successor of x_i and let β be the input to α. Let X' be the set of input variables that β depends on (i.e. that are present in the sub-circuit rooted at β). Observe that $x_i \notin X'$. Consider the truth-table of the sub-circuit rooted at β and

observe it must be a non-constant function of the variables of X'. If it were constant, then we could replace β with this constant and remove α from the circuit via a gate elimination rule, reducing the size of the circuit and violating optimality. Therefore, there is an assignment of the variables in X' such that if we substitute and simplify, eventually a constant will feed into α that allows us to remove it with a fixing rule. This disconnects x_i from the circuit, which violates Fact 2 since we did not substitute for x_i and thus the the resulting parity function must still depend on it.

(2) Suppose x_i has more than two costly successors. Let α_1, α_2 and α_3 be three of these and without loss of generality assume these are indexed in descending depth. Observe that $(\neg)\alpha_3$ is not the output of the circuit as otherwise we could substitute x_i to be some constant that fixes α_3 and make the circuit constant. This would violate the rate limit on eliminations for optimal XOR circuits (Corollary 34). Let β_3 be the costly successor of α_3 and notice that, since α_1, α_2 and α_3 are in descending depth order, β_3 is a new distinct gate. Thus if we substitute x_i to fix β_3 and simplify, we can remove $\alpha_1, \alpha_2, \alpha_3$ and β_3 with a passing or fixing rule applying to β_3 after applying a fixing rule to β_3. This violates Corollary 34.

Both cases reach a contradiction and therefore every input has two costly successors in any normalized optimal circuit computing $(\neg)\mathsf{XOR}_n$. ◀

E.2 Optimal $(\neg)\mathsf{XOR}_n$ Circuits Are Binary Trees of $(\neg)\mathsf{XOR}_2$ Blocks

We now have the tools required to show that binary trees of optimal $(\neg)\mathsf{XOR}_2$ subcircuits are the *only* optimal circuits for computing XOR_n. Formally,

▶ **Theorem 37.** *Optimal* $(\neg)\mathsf{XOR}$ *circuits* partition into *trees of* $(\neg)\mathsf{XOR}_2$ *sub-circuits — even when NOT gates are free. Formally, for every circuit C with the minimum number of AND,OR gates computing* XOR_n*, there is a partition of the gates of C into $(n-1)$ blocks together with a multi-labelling of each wire w in C by tuples $\langle i, t\rangle$ where $t \in \{\mathsf{in}, \mathsf{out}, \mathsf{core}\}$ describes the role that w plays in block i, such that:*

1. *Each block is a three-gate* XOR_2 *sub-circuit, with distinguished* input, output, *and* core *wires.*
2. *The* input *wires of every block are also the output wires of a different block or the input gates.*
3. *Contracting all the* core *wires of C results in a binary tree.*

The proof will proceed via induction, however most of the work will be in proving that we can find a $(\neg)\mathsf{XOR}_2$ block in any XOR_n circuit which we can "peel off" with a single variable substitution. The resulting circuit will compute XOR_{n-1} allowing us apply our inductive hypothesis in order to get a partition we can lift back up to the original circuit. For this reason we separate this out as a lemma.

▶ **Lemma 38.** *Let C be a circuit computing* XOR_n *for $n \geq 3$. There exists two inputs x_i and x_j that feed into a block B in C as described in Theorem 37.*

Proof. Let C be an optimal normalized circuit computing XOR_n where $n \geq 3$. We first identify two variables which will be inputs to block B (which we will later prove that $B \equiv (\neg)\mathsf{XOR}_2$). Let α be a maximum depth binary gate of C. As in the proof of Theorem 31, we know that α must read two distinct inputs x_i and x_j for some $i, j \in [n]$ since otherwise we can apply a simplification rule and remove α, which contradicts that it is an optimal normalized circuit. Our local view of α can be seen in Figure 9, where dashed arrows indicate one (or more) adjacent binary gates whose existence has not yet been established.

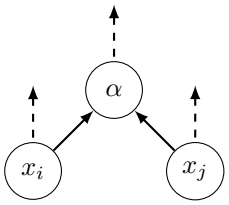


Figure 9 The local neighborhood of α

By Lemma 35, we know that both x_i and x_j have exactly two costly successors. Let β_i be the other successor of x_i. We first observe that neither α nor β_i can be the output of the circuit: if they were, some substitution of x_i would fix the output and leave the resulting circuit constant, violating Corollary 34. Hence, α and β_i must each feed into at least one more costly gate. Let ν be a costly gate fed by α. We remark that ν could be β_i and hence in Figure 10, we denote ν with a dashed node until we establish $\nu \neq \beta_i$.

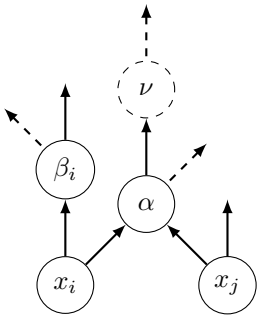
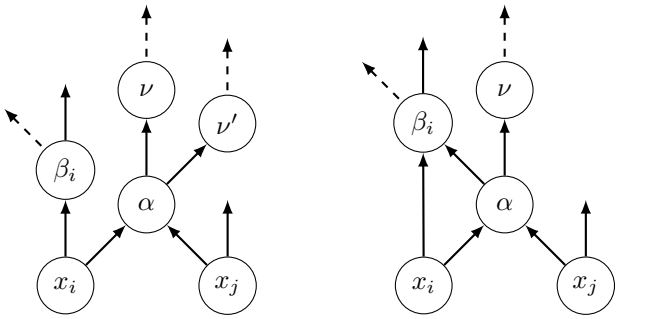


Figure 10 Our view after establishing $\beta_i \neq \alpha$ and neither are the output

We first show that ν is the only successor of α. Towards a contradiction, suppose α also feeds other gates besides ν. If it feeds more than one other gate, then substituting x_i to fix α immediately eliminates more than three gates. Hence, it can at most feed one more gate ν'. In particular, we must analyze two cases depending on whether or not β_i is distinct from ν and ν'. These cases can be seen in Figure 11. In both cases, we can eliminate more than 3 costly gates, violating Corollary 34.

1. Assume β_i is distinct from ν and ν'. Fixing α with x_i will eliminate four distinct gates, $\alpha, \beta_i, \nu, \nu'$.
2. Assume β_i is not distinct from ν and ν'. Without loss of generality, assume $\beta_i = \nu'$. Substituting x_i to fix α also fixes β_i since both of its inputs are now constants. If β_i has any costly successors besides ν then we are done, since that eliminates at least four gates (α, ν, β_i, and any distinct successor). If β_i only feeds into ν, then ν also becomes fixed. Hence, ν cannot be the output of the circuit and thus it must have at least one costly successor that is distinct from α, ν and β_i. This successor can also be eliminated from our substitution.

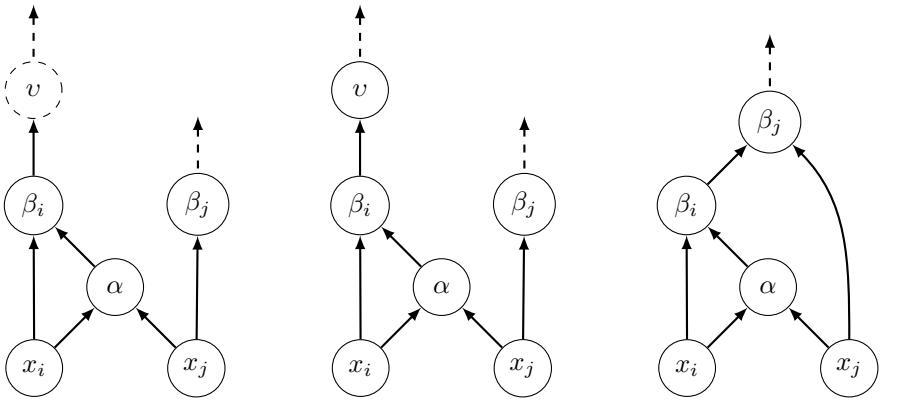
We now show that ν and β_i are distinct. Assume otherwise, then fixing α by substituting x_i also fixes $\beta_i = \nu$, which we know is not the output of the circuit. This eliminates at least 3 (and therefore exactly three) gates (α, β_i, and β_i's successor which we will call υ) and results



(a) β_i is distinct from ν' (b) β_i is not distinct from ν'

Figure 11 The two cases if α feeds two gates

in an optimal XOR_{n-1} circuit. We now consider the other gate fed by x_j which we will call β_j. Again, there are two cases as seen in Figure 12: either β_j is distinct from υ or they are the same.



(a) β_j is distinct from υ (b) β_j is not distinct from υ

Figure 12 The two cases if $\nu = \beta_i$

If β_i does not also feed into β_j, then x_j feeds only β_j in the resulting circuit contradicting Lemma 35. If β_j is fed by β_i, then consider instead substituting x_j to fix β_j. Observe that β_j is not the output of the circuit, and hence fixing it eliminates four gates: α, β_i, β_j, and β_j's costly successors. In both cases we reach a contradiction, hence $\beta_i \neq \nu$. Furthermore, notice that β_i cannot be the output gate and it must have only one costly successor. Otherwise, using x_i to fix β_i would eliminate at least four gates: α, β_i, and the costly successors of β_i. Hence, we arrive at the local view around h as shown in Figure 13.

Now, if we fix α using x_i, then α, β_i, and ν are eliminated and thus must be the only gate that are eliminated. Thus, if β_j, the other successor of x_j is distinct from these three gates, then x_j is left with one costly successor in the resulting circuit computing XOR_{n-1} thus violating Lemma 35. Hence β_j must be one of β_i or ν. We will show it must be β_i.

Suppose, for the sake of contradiction, that $\beta_j = \nu$. We can see that β_i must be fed by ν, otherwise we could substitute x_j to fix α and the resulting XOR_{n-1} circuit only reads x_i once. Now that β_i is fed by ν, if we substitute x_i to fix β_i, then this eliminates β_i, α and ν

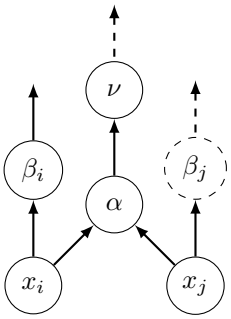


Figure 13 Our view after establishing $\nu \neq \beta_i$

and disconnects x_j entirely from the resulting circuit, but the resulting $(\neg)\mathsf{XOR}_{n-1}$ circuit must still depend on x_j.

We now have that $\beta_j = \beta_i$—let us call the gate β. Recall β has exactly one costly successor, and if this costly successor isn't ν and is another gate then fixing α with x_j eliminates three gates (α, β, ν) but leaves x_j with exactly one costly successor it inherited from β. This is a contradiction, and hence β must feed ν which yields a block B as shown in Figure 14. ◀

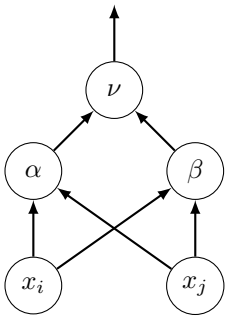


Figure 14 The local area around x_i and x_j form a block B.

We now provide the proof of Theorem 37

Proof of Theorem 37 using Lemma 38. We prove this via induction. For $n = 1$ and $n = 2$ the theorem trivially holds: $(\neg)x_i$ is the unique normal optimal circuit for $(\neg)\mathsf{XOR}_1$ and normal optimal $(\neg)\mathsf{XOR}_2$ circuits trivially define a single block.

Assume the statement holds for some $k - 1 \geq 2$. Let C be a normal optimal circuit computing $(\neg)\mathsf{XOR}_k$ for $k \geq 3$. By Lemma 38, we know C must have a block B that is fed by two distinct inputs $(\neg)x_i$ and $(\neg)x_j$. Let the three gates be α, β, and ν where $(\neg)\nu$ is the output gate of B. We label the wires from $(\neg)x_i$ and $(\neg)x_j$ as `in`, the wires from $(\neg)\nu$ as `out` and all other internal wires of B as `core`. We first need to partition the rest of the circuit into blocks and ensure that the two `out` wires are `in` wires to the same block in the rest of C.

If we substitute $x_i = b$ to fix α and simplify, we see that x_j's successors are replaced by $(\neg)\nu$'s successors and that the three costly gates in B have been eliminated. Therefore the circuit computes $(\neg)\mathsf{XOR}_{k-1}$ and is optimal. Applying the inductive hypothesis, we can

partition the remaining circuit into blocks which we lift back to the original. Notice that x_j's new successors are in the same block and therefore in C, $(\neg)\nu$'s successors are also in the same block as desired.

It remains to prove that B computes $(\neg)\mathsf{XOR}_2$. If we substitute $x_i = b$ to fix h and rewrite as above we see that B reduces to $(\neg)x_j$, i.e. $B(b, x_j) = (\neg)x_j$. We argue that if we instead substitute $x_i = 1 - b$, then B would also reduce to $(\neg)x_j$. It cannot reduce to a constant as otherwise C, which now computes XOR_{k-1} does not depend on x_j, contradicting Fact 2. We also observe B cannot reduce to just x_j in both cases (or $\neg x_j$ in both cases), as otherwise we could replace B with x_j (or $\neg x_j$) and C would still correctly compute XOR_n with three fewer gates — contradicting that C is optimal. Therefore $B(1 - b, x_j) = \neg B(b, x_j)$ and the only binary Boolean functions that satisfy these two equations are XOR_2 and $\neg\mathsf{XOR}_2$. ◀