Finding Circuits for Simple XOR Extensions in Polynomial Time

Tim Jackman

Joint work with Marco Carmosino (IBM) and Ngu (Nathan) Dang

Special Thanks to Rahul Ilango (MIT)

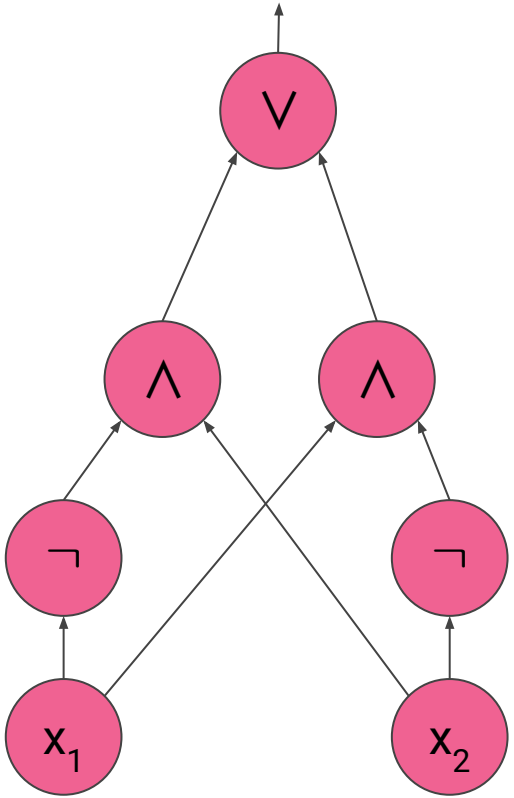
Background & Motivation

Or, What Does The Title Mean?

Circuits

- Used for studying the complexity of Boolean functions f: $\{0,1\}^n$ -> {0,1}
- Circuits are Directed Acyclic Graphs (DAGs) with:
  - 1 sink - the output
  - n sources/leaves - the inputs
- The inputs are labeled $x_1, x_2, \ldots, x_n$
- Interior nodes, called *gates*, are labeled by Boolean functions from a **Basis** set
  - Example: $\{\wedge, \vee, \neg\}, \{\oplus, \wedge\}$
- DeMorgan Basis: $\{\wedge, \vee, \neg\}$
  - $\wedge, \vee$ have fanin 2, $\neg$ has fanin 1
  - Unbounded fanout on all gates
- Normalization:
  - No double negations, all gates feed into at most one $\neg$ gate

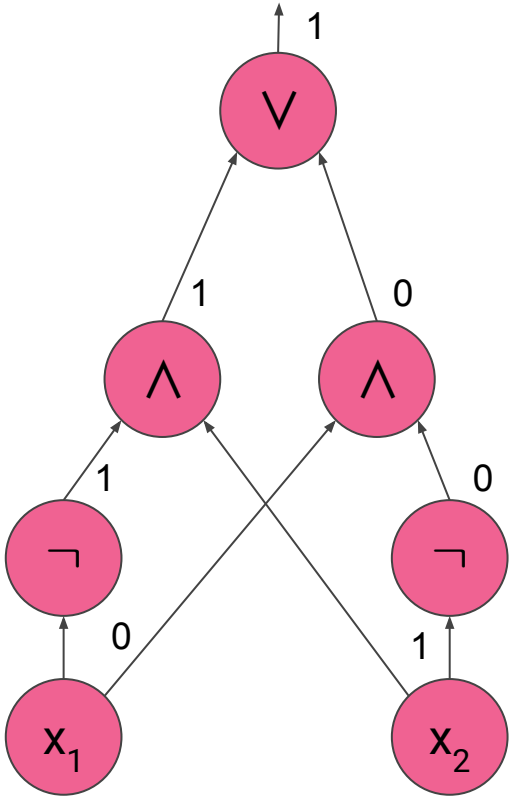
An Example Circuit



- Computes XOR_2

x_1	x_2	$XOR_2(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

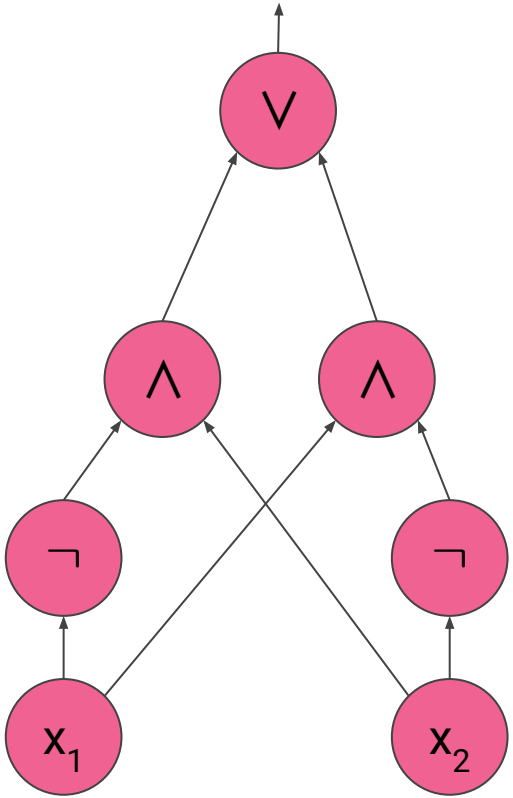
An Example Circuit



- Computes XOR_2

x_1	x_2	$XOR_2(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

An Example Circuit



- Computes XOR_2
- Complexity Measure:
  - Depth
  - Size
    - Number of $\wedge$, $\vee$ gates
    - $\neg$ gates are free
- This circuit has size 3
- CC(f) is the size of the smallest circuit computing f

The (in)famous Minimum Circuit Size Problem (MCSP)

Input: $f : \{0,1\}^n \rightarrow \{0,1\}$ as a truth table (2^n bitstring) & $s \in \mathbb{N}$

Output: Whether $CC(f) \leq s$

- In NP and…
- …NP-completeness is open…
- …few variants are known to be NP-complete [Mas 79, HOS 18, ILO 20, Ila 20]
- Resolving the question would imply major breakthroughs
  - If MCSP is NP Complete then EXP ≠ ZPP [MW15]
  - If MCSP $\in$ P then there are no one way functions [KC00]
  - And many many many more…

What do we even know about circuit complexity?

Taking a Step Back

- Most Boolean functions require large ($\Omega(2^n/n)$) circuits [Sha 49]
- Known DeMorgan Circuit Lower Bounds (for Functions in NP)
  - $CC(XOR_n) = 3(n-1)$ [Schnorr 1973]
  - $CC(MOD_4) \geq 4(n-7)$ [Zwick 1991]
  - $CC(\text{“k-mixed”}) \geq 4.5n - o(n)$ [Lachish and Raz, 2010]
  - $CC(WSUMP) \geq 5n - o(n)$ [Amano and Tarui 2011]
- 5n upper bound for the best lower bounds [AT11]
- Can we do better with more resources (non-determinism, randomness)?
  - MAEXP does not have polynomial size circuits [BFT 98]
  - $ZPEXP^{MCSP}$ does not have polynomial size circuits [IKV 18]
  - MA/1 is not in $SIZE[n^k]$ for any k [Santhanam 19]

Relative Circuit Complexity

- Proving circuit lower bounds seems to require a lot of work
- How do these lower bounds lift to extensions?
  - $g : \{0,1\}^{n+m} \to \{0,1\}$ is an **extension** of $f : \{0,1\}^n \to \{0,1\}$ if

$$\exists\ k \in \{0,1\}^m\ \forall\ x \in \{0,1\}^n : g(x,k) = f(x)$$

- If g is an extension of f then $CC(g) \geq CC(f)$
  - Substitute in k in any g circuit to get an f circuit
- g is a **non-degenerate** extension of f
  - $\forall\ i \in [n+m]\ \exists\ x \in \{0,1\}^{n+m} : g(x) \neq g(x^{\oplus i})$
  - Extension variables must be read in g circuits
  - $CC(g) \geq CC(f) + m$

The $\mathscr{F}$- Simple Extension Problem

Let $\mathscr{F} = \{f_n\}_{n \in \mathbb{N}}$ be a sequence of non-degenerate Boolean functions on n variables

Input: $g : \{0,1\}^{n+m} \rightarrow \{0,1\}$ represented as a truth table, $n \in \mathbb{N}$

Output: Whether g is a non-degenerate extension of f_n and $CC(g) = CC(f_n) + m$

- Example:
  - OR_7 is a simple extension of OR_3
- Non Examples:
  - XOR_6 is **not** a simple extension of XOR_5

The $\mathscr{F}$- Simple Extension Problem

Let $\mathscr{F} = \{f_n\}_{n \in \mathbb{N}}$ be a sequence of non-degenerate Boolean functions on n variables

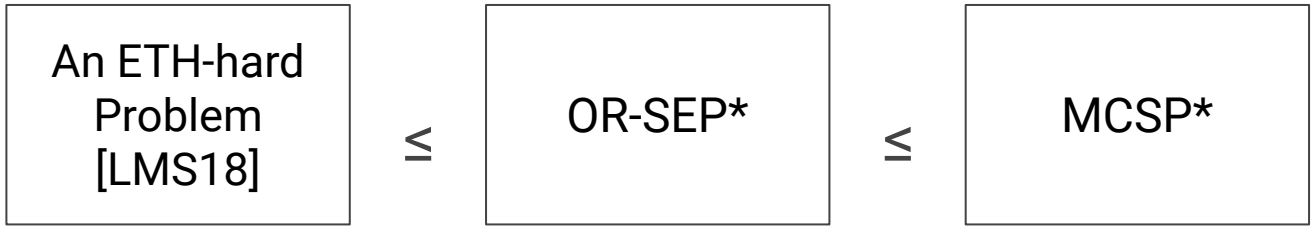
Input: $g : \{0,1\}^{n+m} \to \{0,1\}$ represented as a truth table, $n \in \mathbb{N}$

Output: Whether g is a non-degenerate extension of f_n and $CC(g) = CC(f_n) + m$

- Like MCSP, this is in NP
- Checking that g is a non-degenerate extension of f_n is “easy”
- Checking $CC(g) = CC(f_n) + m$ reduces to MCSP
- Hardness of $\mathscr{F}$-SEP implies hardness for MCSP

Partial Function MCSP (MCSP*) Is ETH Hard

- MCSP* is MCSP but with *partial* truth-tables:
  - Is there any completion of the truth-table whose CC is at most k?
- MCSP* is hard assuming the Exponential Time Hypothesis (ETH) [Ila 20]
  - ETH : SAT cannot be solved in subexponential time
  - Proved via a reduction from Partial OR-Simple Extension



Awesome! Why don't we just lift this proof to ETH hardness of full MCSP?

The Catch

- Identifying simple extensions of **total** functions whose optimal circuits are read-once formulas (ROF) is *easy*.
  - **Read-once formula** ≔ fanout of every node is 1
  - *f* is computed by a ROF ⇒ CC(f) = n - 1
  - g is a simple extension of f ⇒ g's optimal circuits are ROFs
  - Deciding if functions can be computed by ROFs is in P [AHK93, GMR06]
- Ilango's proof used structural knowledge of OR

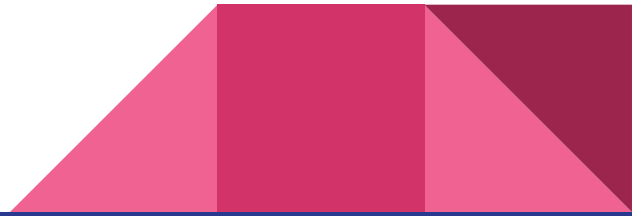
"… the missing component in extending our results to MCSP is finding some function f whose optimal circuits we can characterize but are also sufficiently complex."

Rahul Ilango, SIAM J. of Computing, 2022

XOR : The Next Natural Candidate

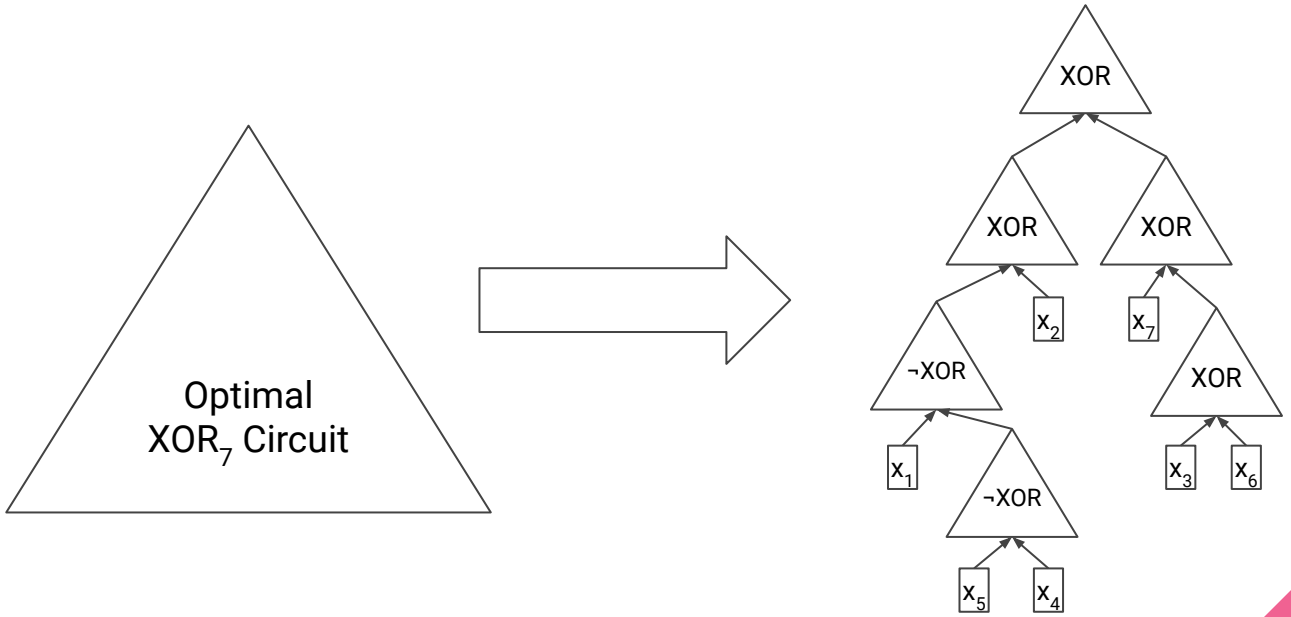
- Not computed by ROFs
- DeMorgan Basis Circuit Complexity is *exactly* known
  - 3(n - 1) Lower Bound from Schnorr
  - 3(n - 1) Upper Bound by Composing XOR_2 subcircuits
- We've fully characterized the set of optimal XOR_n circuits:

  <u>All</u> optimal XOR_n circuits are binary trees of (¬)XOR_2 subcircuits



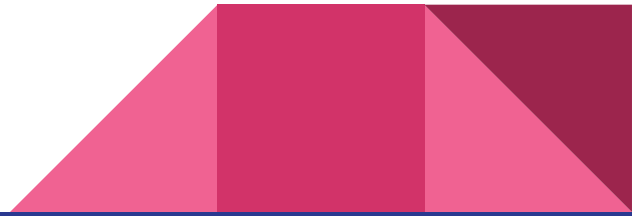
XOR : The Next Natural Candidate

Theorem: <u>All</u> optimal XOR_n circuits are binary trees of (¬)XOR_2 subcircuits

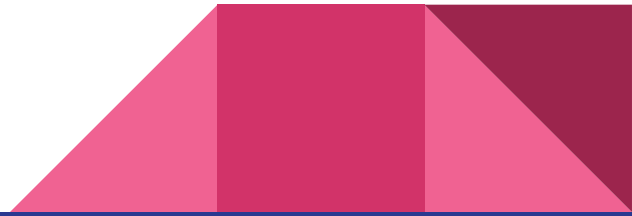


Awesome, Let’s Try to Prove MCSP is ETH Hard Using XOR Simple Extension

Bad News



~~Bad News~~ Good News?



~~Bad News~~ ~~Good News?~~ News

Main Theorem: XOR_n-Simple Extension is in P

- We design a “generic” algorithm for f-Simple Extension
  - Running time depends on “shape” parameters of optimal circuits
    - Is not polynomial time in general
  - For XOR, it is polynomial
    - Probably also polynomial for the other explicit functions with lower bounds

Naive Brute Force

Input: tt(g) (2^{n+m} bit string)

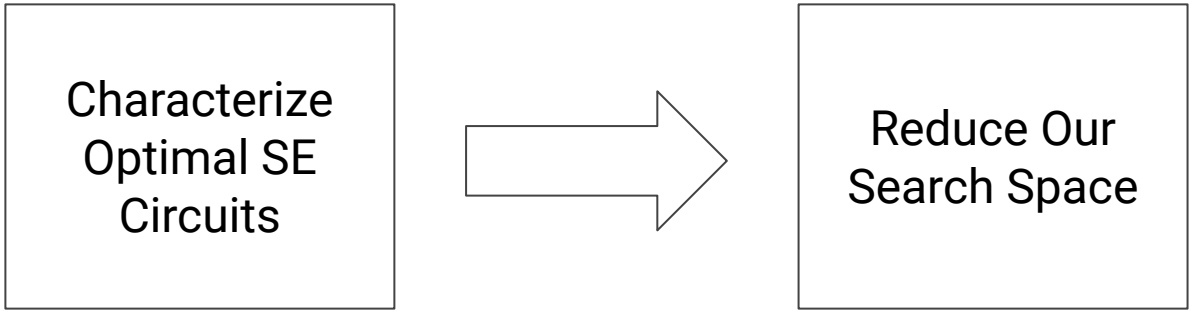
Output: Whether g is a non-degenerate extension of f with CC(g) = CC(f) + m

- Brute-force checking for a key & non-degeneracy is “efficient”
- Suffices to find a circuit of size CC(f) + m
- Just check all circuits of the appropriate size:
  - Encoding argument: a circuit of size s can be encoded in O(s log s) bits
    - ≈ $2^{O(s \log s)}$ circuits
  - CC(f) is Ω(n) - there’s $2^{\Omega((n+m)\log(n+m))}$ circuits to check
    - Quasipolynomial time

We’ll just need to be clever then…

But I Really *Really* ***REALLY*** Like Brute Forcing Things

- Fine, but we need to come up with something better to brute force over.

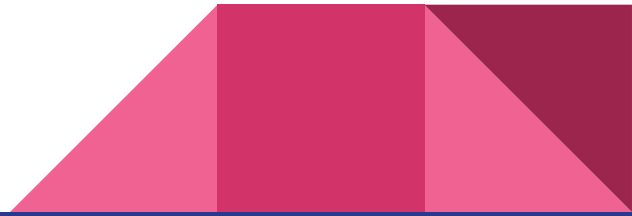


Building Up Our Toolbox

How can we design an algorithm if we don’t know anything?

Notation & Basic Tools

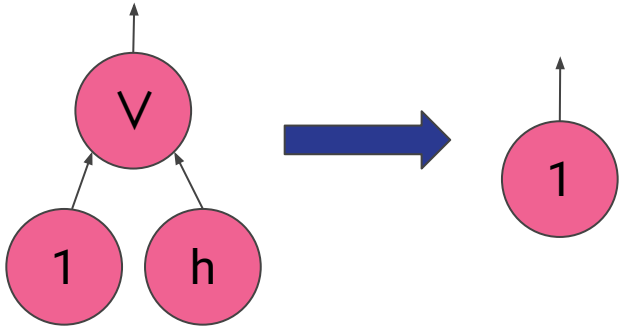
- For a simple extension g, we separate its' inputs into:
  - "Original" variables: $x_1, x_2, \ldots, x_n$
  - "Extension" variables: $y_1, y_2, \ldots, y_m$
- We refer to restrictions $k \in \{0,1\}^m$ such that $g(x, k) = f(x)$ as ***keys***
- We refer to AND and OR gates as **costly** gates
- **Main Tool**: Substitution with keys in circuits and performing **gate elimination**



Gate Elimination

- Circuit simplification scheme which removes constants & “obvious” inefficiencies in a circuit

“Fixing” Rules: Removed gates become “fixed” constants

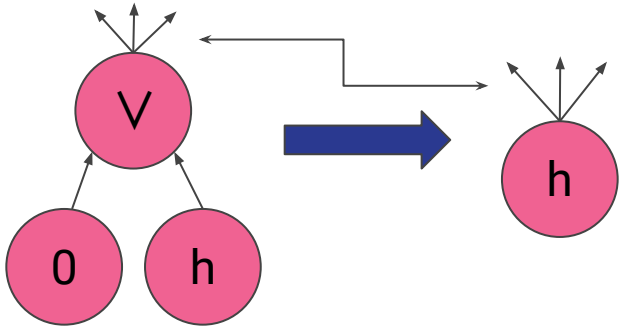


Fixing rules are hard to analyze since more simplifications must occur

Gate Elimination

- Circuit simplification scheme which removes constants & “obvious” inefficiencies in a circuit

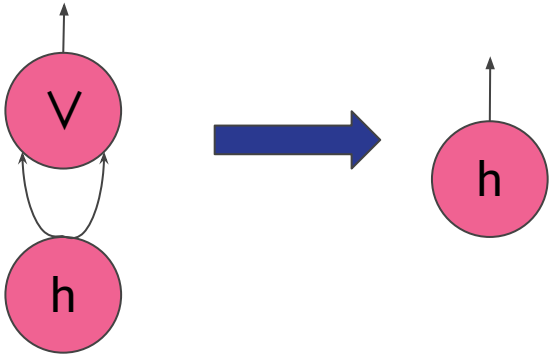
“Passing” Rules: Removed gates “pass” wires to their other input



Passing rules are easier to analyze since they remove constants

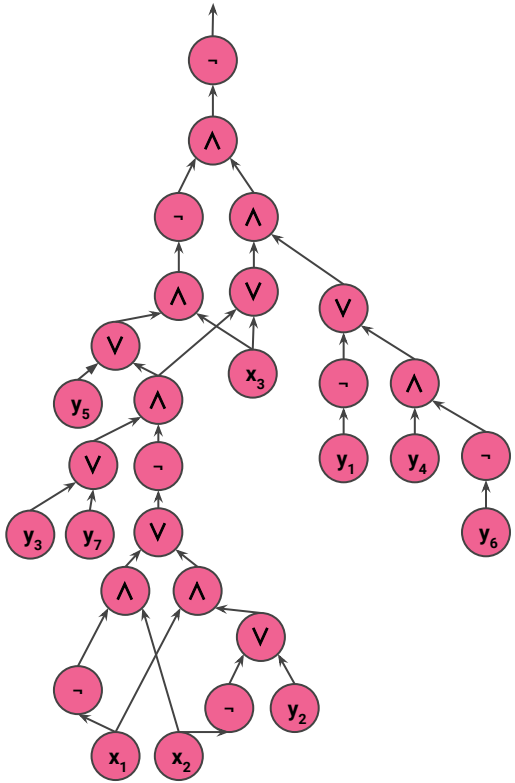
Gate Elimination

- Circuit simplification scheme which removes constants & “obvious” inefficiencies in a circuit



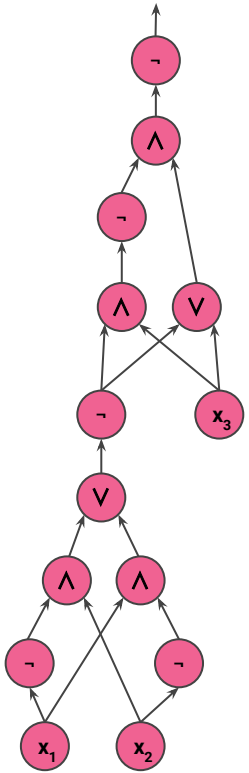
Arguments using gate elimination are a case analysis nightmare

An Example of a Simple Extension Circuit



- This is an optimal circuit for a simple extension of XOR_3 with 7 extension variables
- If we restrict with k = 1001001 we get…

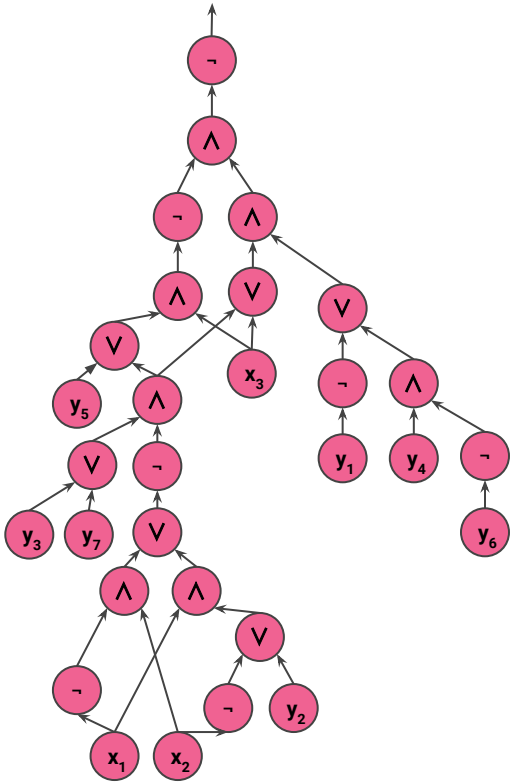
An Example of a Simple Extension Circuit



- This is an optimal circuit for a simple extension of XOR_3 with 7 extension variables
- If we restrict with k = 1001001 we get…

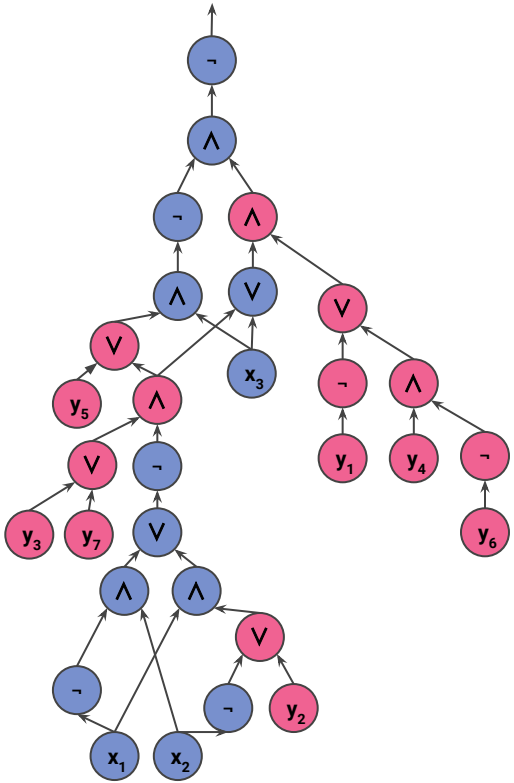
An optimal XOR_3 circuit!

An Example of a Simple Extension Circuit



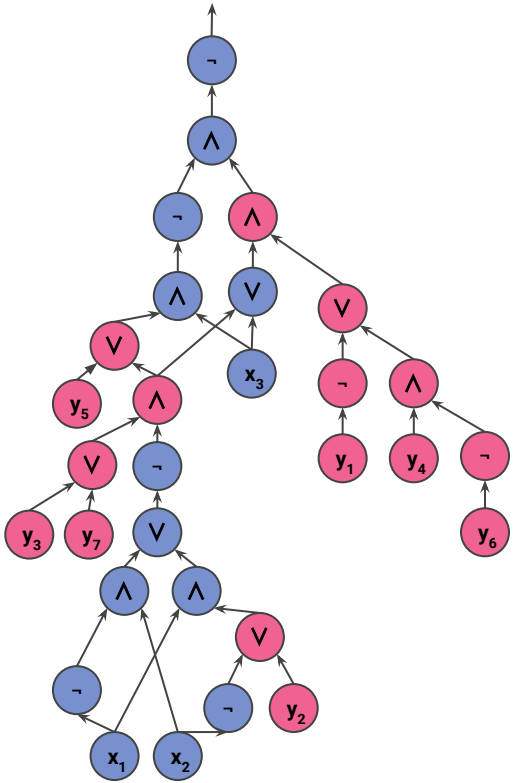
- Let’s highlight this embedded circuit…

An Example of a Simple Extension Circuit



- Let’s highlight this embedded circuit…

An Example of a Simple Extension Circuit

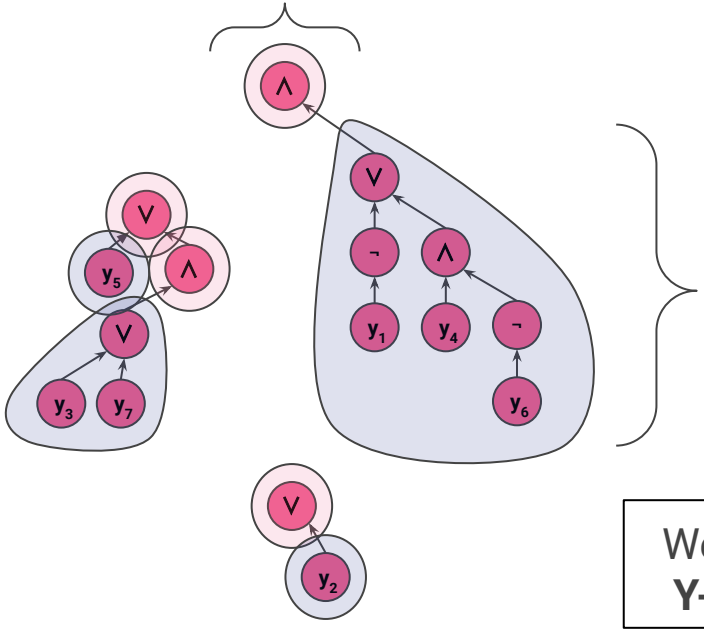


- Let’s highlight this embedded circuit…

  …and focus in on what’s added on to it

An Example of a Simple Extension Circuit

An added gate that ***combines*** the tree with the rest of the circuit



- Let's highlight this embedded circuit…

…and focus in on what's added on to it

Full **Tree** which only reads **y variables.** Furthermore, it's a *formula*

We call these structures **Y-trees** and **combiners**

Structural Claims

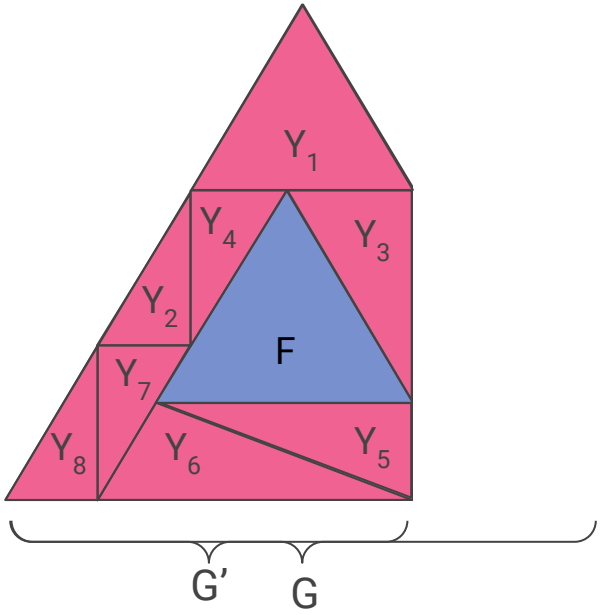
- These structures are not unique to our example
- **Embedding Lemma**: Every optimal circuit for a SE has an embedded optimal circuit for the base function
  - Substituting m non-degenerate variables & simplifying reduces circuit size by at least m
- **Structural Theorem:** All optimal SE circuits can be decomposed into:
  - The embedded base circuit
  - Non-intersecting Y-trees & their respective “combiners”
- **Completely generic - nothing to do with XOR**

Proof Sketch of Structural Theorem

- Relies on the following lemmas:
  - **Intermediate SE Lemma:** Restricting G with a partial key produces an optimal circuit for an intermediate simple extension lying between f & g
    - Restricting s < m extension variables may eliminate > s costly gates due to fixing rules
  - **Good Keys Lemma:** For any s extension variables, there's some partial key restriction of those variables that eliminates exactly s costly gates.
    - Good key substitution for the variables in a Y tree eliminates just the Y-tree + the combiner
- We will prove this via induction on m

Proof Sketch Continued

- Take any optimal g circuit G
- There is an f circuit F embedded in G
- Identify a Y-tree & combiner
- Eliminate it using a *good* key
- Resulting G’ is optimal for intermediate SE by lemma
- Apply Inductive Hypothesis
- Lift decomposition back to G



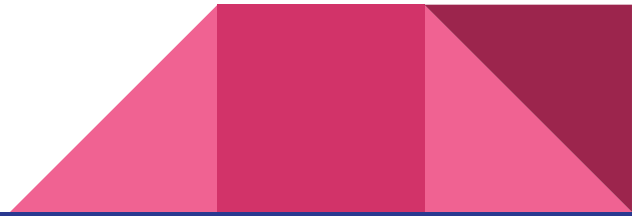
Our “Efficient” Algorithm

Step Aside Brat Summer It’s Brute-Force Fall

Our Strategy

- Every optimal SE circuit is an embedded optimal base circuit + Y-trees
- Take optimal base circuits & build extension ckts by adding Y-trees
- Check if what we built computes g
- Reject once we've exhausted all optimal base circuits

We'll need to make sure our search space is sufficiently small so thus brute force is "efficient"



$2^{O(n+m)}$ is Efficient? I mean…

- Input is a truth table of a Boolean function on n+m variables:
  - 2^{n+m} bits long ⇒ poly(2^{n+m}) ⇒ $2^{O(n+m)}$
- What's Allowed:
  - Going over the truth table *a lot* : $2^{O(n+m)} * 2^{O(n+m)} = 2^{O(n+m)}$
  - Computing truth tables for size s circuits: $O(s * 2^{n+m})$
- What's Not:
  - $2^{O(n \log n)}$
  - **n! - PERMUTATIONS ARE OFF THE TABLE**
    - Rahul's partial hardness relies on searching over n! being unavoidable
- What's kind of allowed?
  - Dependence on other parameters that are small for XOR
    - Circuit size -> O(n)
    - Maximum fanout -> O(1)

Counting is Fun!

To Fix Later:
- # of Base Circuits

- Sanity Check - Is It Feasible For XOR_n?
  - XOR_n circuits are binary trees of (¬)XOR_2 subcircuits:
    - # of unlabeled binary trees on n inputs:
      - $(n-1)^{th}$ Catalan Number C_{n-1}

      $C_{n-1} = O(4^{n-1}) = 2^{O(n)}$

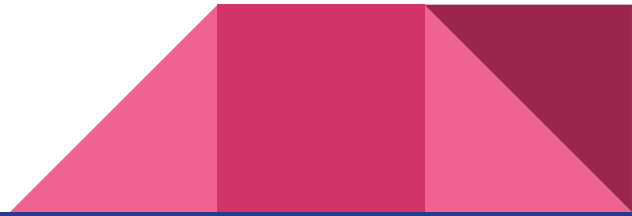
      ng the inputs

      ptions

Counting is Fun

To Fix Later:
- # of Base Circuits

- We can imagine “splicing” in combiners & Y-trees “one-by-one”
- How many Y-trees are there? How many ways to splice in Y-trees?



Counting is Fun

To Fix Later:
- # of Base Circuits

- **How many Y-trees are there?** How many w[illegible]e in Y-trees?
- # of extension variables in a Y-tree ranges from 1 to m
  - *d* extension variables
- Y-trees are Boolean formulas ≈ Weighted Binary Trees
  - Interior nodes labeled AND/OR
  - Negations correspond to edges with weight 1
- # of unweighted BT w/ d leaves = $C_{d-1} = 2^{O(d)}$
- # of edges = 2(d-1) + 1 => $2^{O(d)}$ options for weights
- Labeling inputs => d!

Counting is Fun

To Fix Later:
- # of Base Circuits
- # of Y-trees

- **How many Y-trees are there?** How many w[illegible]e in Y-trees?
- # of extension variables in a Y-tree ranges from 1 to m
  - *d* extension variables
- Y-trees are Boolean formulas ≈ Weighted Binary Trees
  - Interior nodes labeled AND/OR
  - Negations correspond to edges with weight 1
- # of unweighted BT w/ d leaves = $C_{d-1} = 2^{O(d)}$
- # of edges = 2(d-1) + 1 => $2^{O(d)}$ options for weights
- Labeling inputs => d!

Counting is “Fun”

To Fix Later:
- # of Base Circuits
- # of Y-trees

- **How many Y-trees are there?** How many w[illegible]e in Y-trees?
- When there are multiple Y-trees we need to divy up the extension variables
  - We are partitioning the set $\{y_1, y_2, \ldots, y_m\}$ into t subsets
  - $\sum_{1 \le t \le m}$ (# of ways to partition an m-set into t subsets)
    - Equals the m^{th} Bell Number B_m
    - $B_m \ge (m/2)^{(m/2)}$

Counting is “Fun”

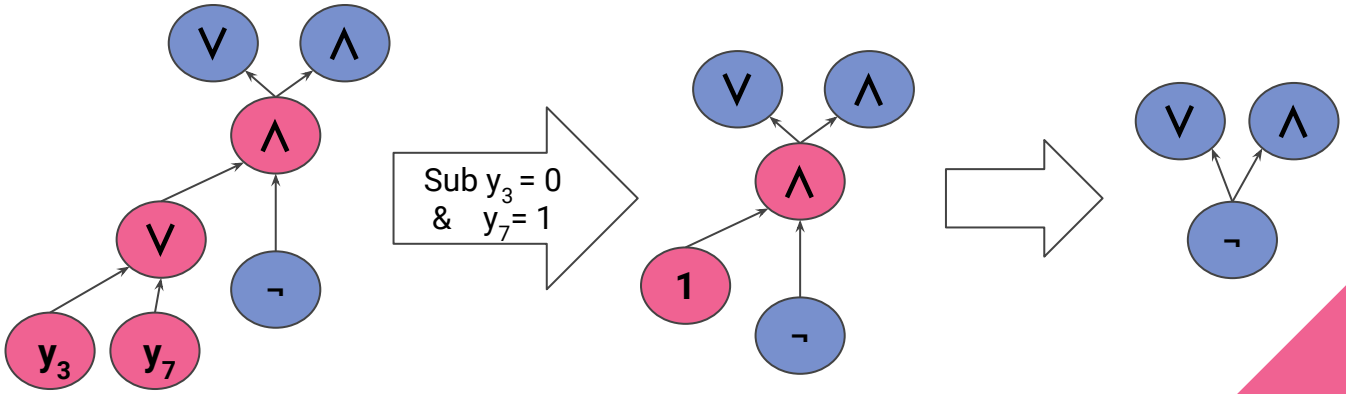
To Fix Later:
- # of Base Circuits
- # of Y-trees
- Partitioning y’s

- **How many Y-trees are there?** How many w[illegible]e in Y-trees?
- When there are multiple Y-trees we need to divy up the extension variables
  - We are partitioning the set $\{y_1, y_2, \ldots, y_m\}$ into t subsets
  - $\sum_{1 \le t \le m}$ (# of ways to partition an m-set into t subsets)
    - Equals the m^{th} Bell Number B_m
    - $B_m \ge (m/2)^{(m/2)}$

Counting is “Fun”

To Fix Later:
- # of Base Circuits
- # of Y-trees
- Partitioning y’s

- **How many ways to splice in Y-trees are there?**
- Where can we put combiners?
- Observation: When a combiner is eliminated via “good” keys, it’s out wires are “passed” down to it’s other input



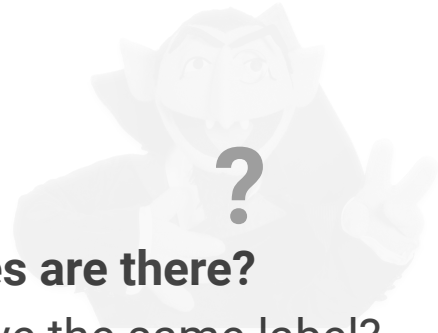
Counting is “Fun”

To Fix Later:
- # of Base Circuits
- # of Y-trees
- Partitioning y’s

- **How many ways to splice in Y-trees are there?**
- Where can we put combiners?
- Observation: When a combiner is eliminated via “good” keys, it’s out wires are “passed” down to it’s other input
  - During splicing we can think of “stealing” some out edges from some “original” node in the original base circuit
  - Label each combiner with what it “steals”
    - O(s) choices for the “origin” where s = CC(f)
    - $2^{O(\text{max-fanout}(F))}$ choices for what to steal

What if two or more combiners have the same label?

Counting is ~~“Fun”~~



To Fix Later:

- # of Base Circuits
- # of Y-trees
- Partitioning y’s

- **How many ways to splice in Y-trees are there?**
- What if two or more combiners have the same label?
  - We have to decide how to order them with respect to one another
  - If we have k Y-trees with the same label…
    - k! ways to arrange them in a stack

Counting is ~~“Fun”~~

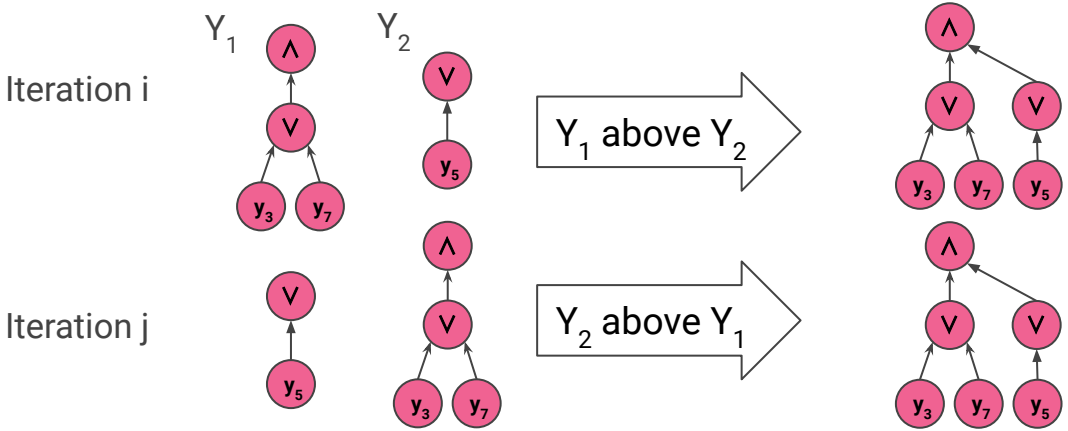
- **How many ways to splice in Y-trees are there?**
- What if two or more combiners have the same label?
  - We have to decide how to order them with respect to one another
  - If we have k Y-trees with the same label…
    - k! ways to arrange them in a stack

To Fix Later:
- # of Base Circuits
- # of Y-trees
- Partitioning y’s
- Ordering Y-Trees

Counting is ~~"Fun"~~

- **How many ways to splice in Y-trees are there?**
- What if two or more combiners have the same label?
  - We have to decide how to order them with respect to one another
  - If we have k Y-trees with the same label…
    - k! ways to arrange them in a stack

To Fix Later:
- # of Base Circuits
- # of Y-trees
- Partitioning y's
- Ordering Y-Trees



Counting is ~~“Fun”~~

To Fix Later:

- # of Base Circuits
- # of Y-trees
- Partitioning y’s
- Ordering Y-Trees

- **How many ways to splice in Y-trees are th**
- What if two or more combiners have the same label?
  - We have to decide how to order them with respect to one another
  - If we have k Y-trees with the same label…
    - k! ways to arrange them in a stack
    - We can just arbitrarily label the added trees Y_1, Y_2, …
    - If Y_i and Y_j have the same label & i > j then Y_i appears above Y_j

Counting is ~~"Fun"~~

To Fix Later:
- # of Base Circuits
- # of Y-trees
- Partitioning y's
- ~~Ordering Y Trees~~

- **How many ways to splice in Y-trees are th**
- What if two or more combiners have the same label?
  - We have to decide how to order them with respect to one another
  - If we have k Y-trees with the same label…
    - k! ways to arrange them in a stack
    - We can just arbitrarily label the added trees Y_1, Y_2, …
    - If Y_i and Y_j have the same label & i > j then Y_i appears above Y_j

No But Seriously… What About The Permutations?

- Permutations pop up in our counting in a few spots:
  - The # of base XOR_n circuits
  - The # of Y-trees
- We also run into issues partitioning the Y variables amongst the Y-trees

Silver Bullet: Truth Table Isomorphism

Well That Was Easy

- Two Boolean functions f & g are ***truth table isomorphic*** if there exists a permutation π: [n] $\rightarrow$ [n] such that g(x) = f(π(x))
  - If f and g are tt-iso then CC(f) = CC(g)
    - Take any f circuit and permute its inputs to get a circuit for g
  - Checking tt-iso is in P [Luks 99]
- Can assign variables arbitrarily
  - Brute force over “open” (i.e. unlabeled input) circuits & subcircuits
- Avoid partitioning the set of extension variables
  - Decide how many Y-trees & how many vars each has
  - # of compositions of m -> 2^{m-1}

Wooo Brute-Force!

On input tt(g), n:

1. Check that g is a non-degenerate extension of f_n
2. For each "open" optimal circuit of f:
   a. For each "open" Y-tree decomposition:
      i. Construct G' := the base circuit w/ the Y-tree decomposition spliced in where variables are filled arbitrarily
      ii. Compute tt(G')
      iii. If tt(G') ≅ tt(G) then **accept**
3. Reject

Running Time: $L_n \cdot 2^{O(\varphi(s+m))}$ where
L_n is the # of open optimal ckts for f_n (up to isomorphism)
φ is the maximum fanout in any optimal ckt for f_n

It works for XOR

- The algorithm is poly time if #OpenOptCkts(f_n) = $2^{O(n)}$, CC(f_n) = O(n), and the maximum fanout in any optimal circuit is constant
- Every optimal XOR_n circuit is a binary tree of (¬)XOR_2 subcircuits
  - C_n unlabeled binary trees
  - Label each internal node with a (¬)XOR_2 subcircuit
    - Some constant number of options, say $\leq 2^8$
    - (n-1) subcircuits to label
  - Total number of “open” XOR_n circuits: $\leq 2^8(n-1)_, C_n = 2^{O(n)}$
  - Maximum fanout is 2

Discussion & Future Directions

Okay? What Now?

What Else Is Ruled Out?

Bypassing our algorithm will require:
1) Better structural knowledge of circuits
2) Significantly improved lower bounds

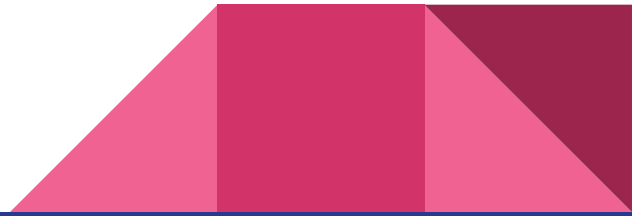
- If a function's optimal circuits are
  - of O(n) size,
  - with constant maximum fanout,
  - and polynomially many (up to permutation of inputs),

  then it is not a candidate for hardness of MCSP
- If the best known constructions for:
  - MOD4
  - WSUMP

  Are optimal & exhaustive then they are ruled out

Tweaking The Model

- The reduction to MCSP still holds even if we:
  - Change the Basis
    - B_2 (all binary Boolean functions)?
  - Increase the “gap”
    - $CC(g) \leq CC(f) + c \cdot m$ for some constant c
- Other models of Boolean computation:
  - Formulas -> MFSP is Hard [Ilango 21]
  - Branching Programs



Do We Need Non-Degeneracy?

- Reduction to MCSP still holds without non-degeneracy
  - Non-degeneracy yields structure
  - Can “pad” the the truth table with degenerate variables increasing gap
- Subtly “easier” than the general relative complexity problem
  - **Input**: Given g an extension of f & a natural number d

    Output: Whether CC(g) = CC(f) + d
  - Padding in the truth-table allows our running time to be longer

Thank you!

Appendix

You wanted to hear more?

Our “Actual” Algorithm

- Actual implementation of brute-forcing:
  - Encode the sequence of Y-trees splices from F to G
  - Design an efficient decoding algorithm
  - Brute force over all possible encodings
- Encoding length we get is max-fanout(F)(s + m)
  - Check all $2^{\text{max-fanout}(F)(s+m)}$ possible encodings
- Same issues that affected our counting occur in encoding
  - Truth-table isomorphism saves us the same bits
- One extra issue - specifying splice locations with names
  - Splice Y-trees in increasing topological order
  - Specify how many more gates up the next origin is