



《设计自动化与计算系统》 Logic Synthesis

The slides are based on Prof. Weikang Qian's lecture notes at SJTU and Prof. Rob Rutenbar's lecture notes at UIUC

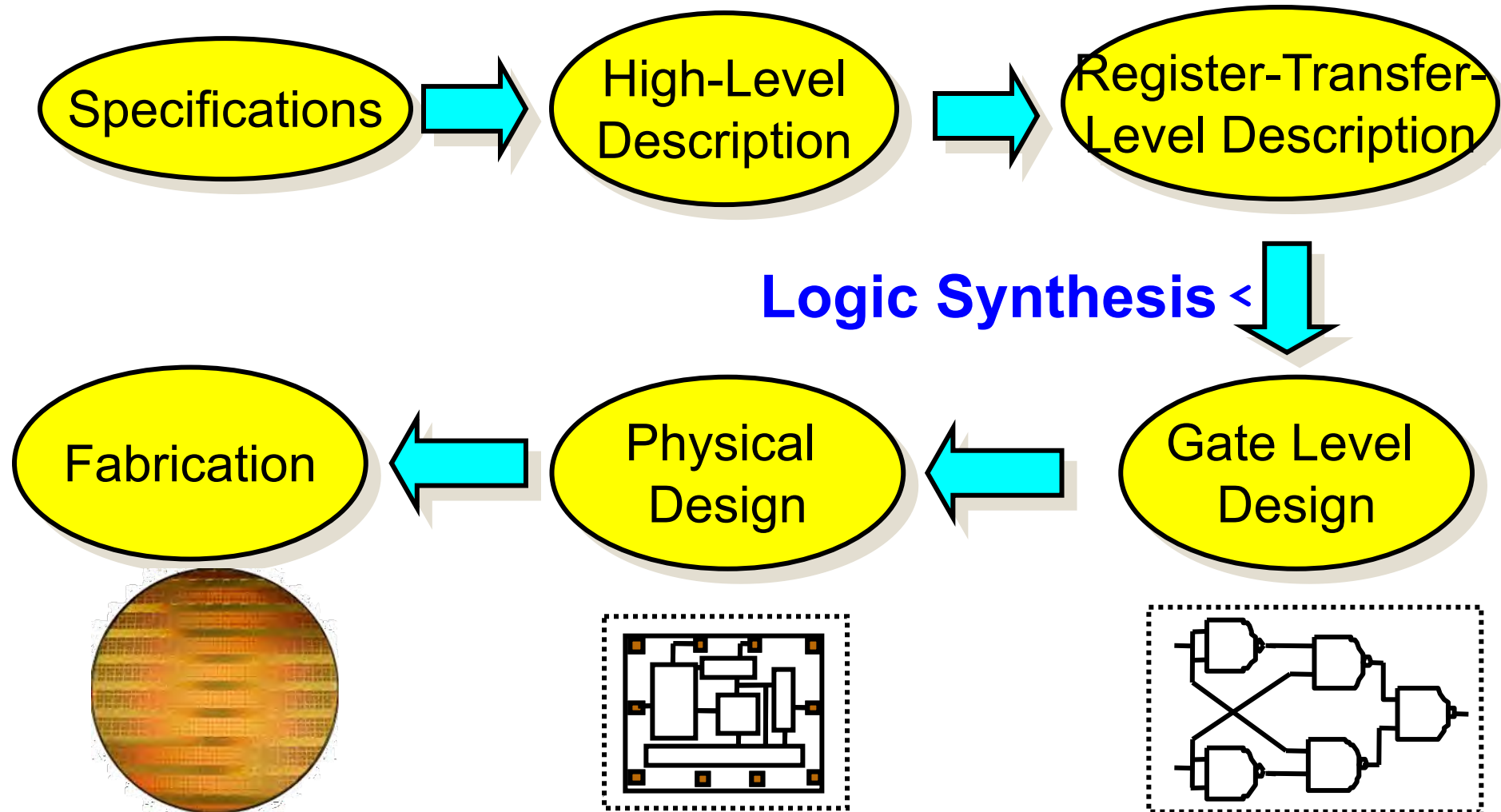
Yibo Lin

Peking University

Outline

- Background on Logic Synthesis
- Technology-Independent Synthesis
- Technology Mapping
- Exact Synthesis

IC Design Steps



Logic Synthesis

RTL design
description



Logic
Synthesis



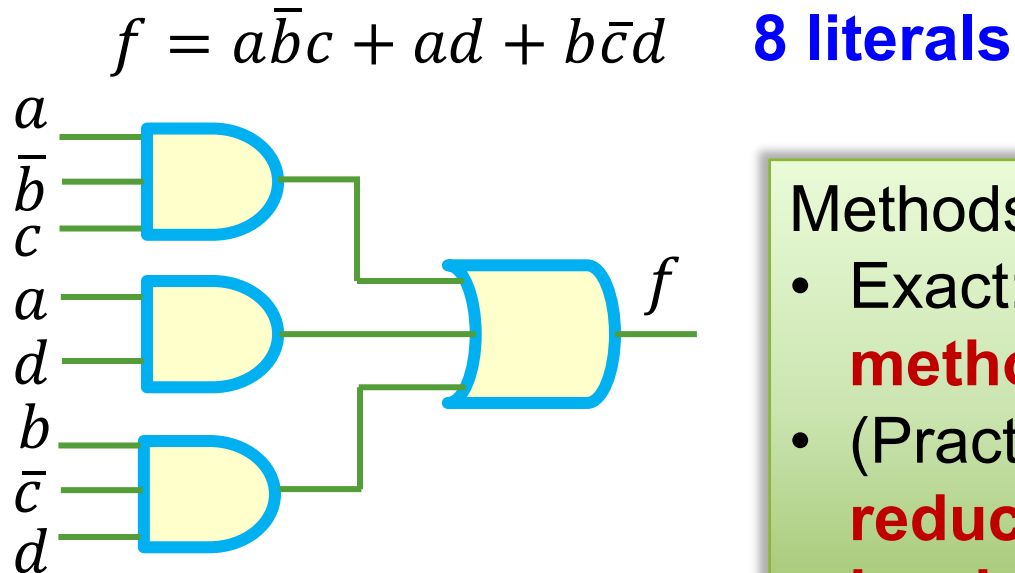
Connected gates + wires
in your technology.
Called “netlist”

➤ Logic design styles

- Two-level logic
- Multi-level logic

Two-Level Logic Design

- A logic implementation based on **sum-of-products (SOP)** expression.
 - First level: many AND gates.
 - Second level: one “big” OR gate.
- Optimization target: minimize **number of literals**

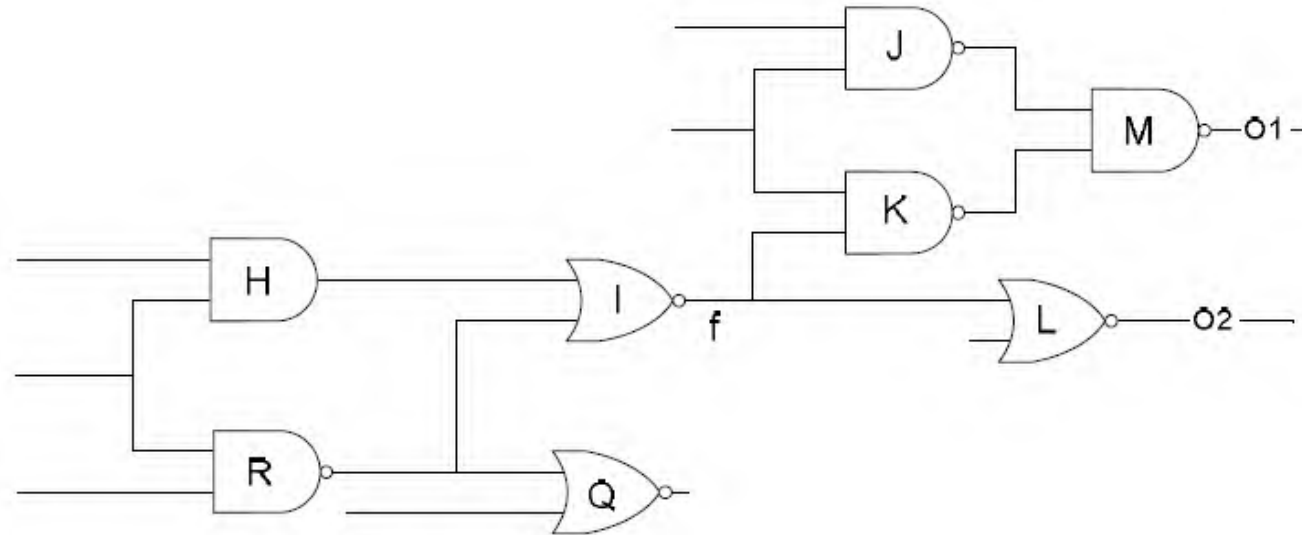


Methods

- Exact: **Quine-McCluskey method**
- (Practical) Heuristic: **reduce-expand-irredundant loop**

Our Focus: Multi-Level Design

- Modern design style!

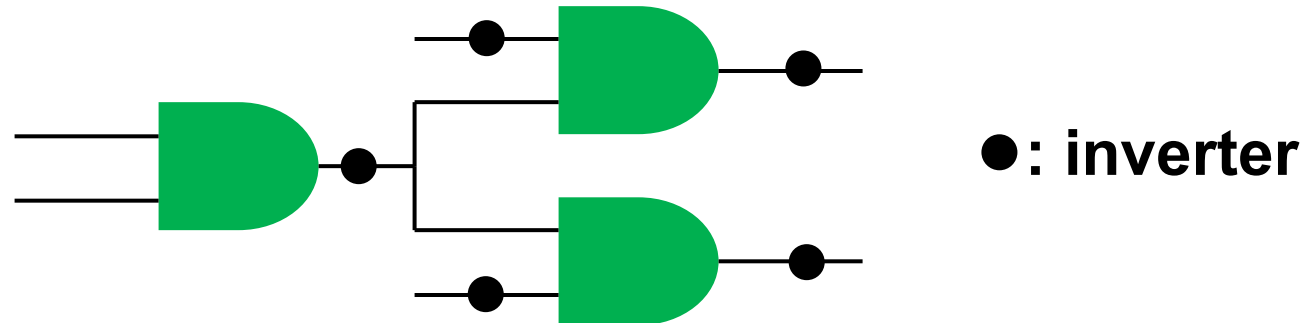


- Two phases in multi-level logic synthesis:
 - Technology-independent synthesis
 - Technology mapping

Technology-Independent Synthesis

- Output: **An abstract representation of Boolean function**
 - It is **not** the actual **gate-level netlist**
 - The result is called: “**technology independent**” logic or “**uncommitted**” logic
 - Example: **AND-inverter graph (AIG)**, **Boolean logic network**, etc.

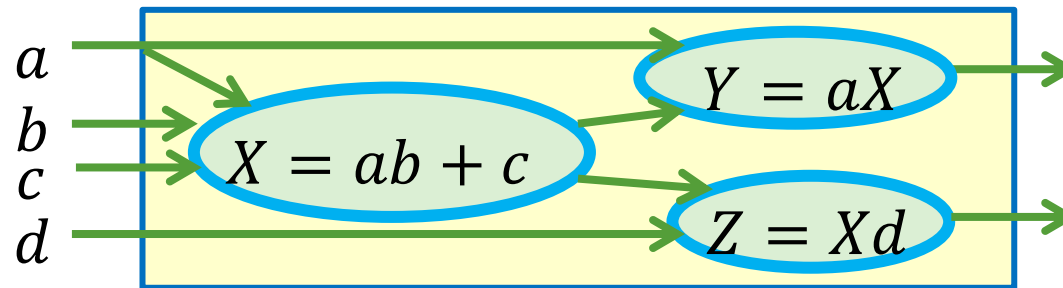
AND-inverter graph (AIG)



Another Example of Abstract Representation : Boolean Logic Network

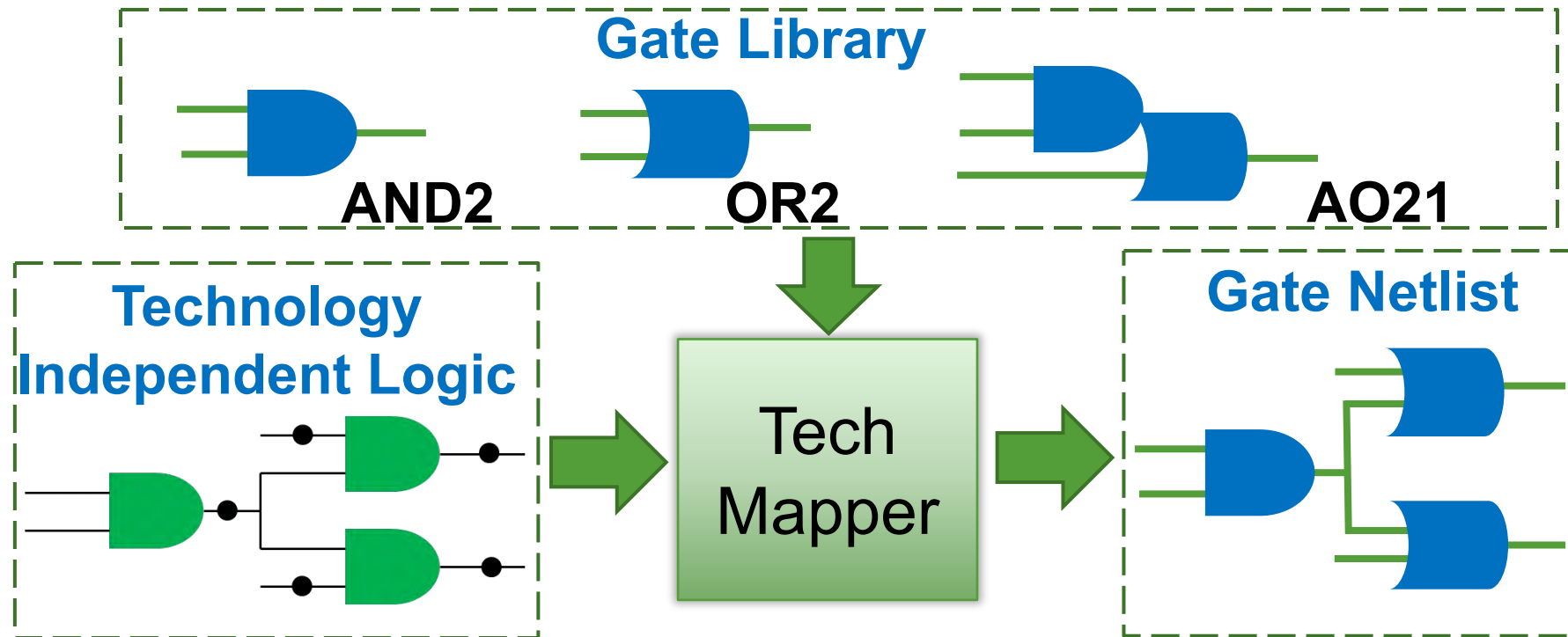
- A graph of connected blocks with each block being a 2-level Boolean functions in **sum-of-product (SOP) form**.

— Our focus



Technology Mapping

- Given a gate library, from which we can pick gates, and **technology independent** logic F , build a final **gate netlist** for F using only these gates from the given library.

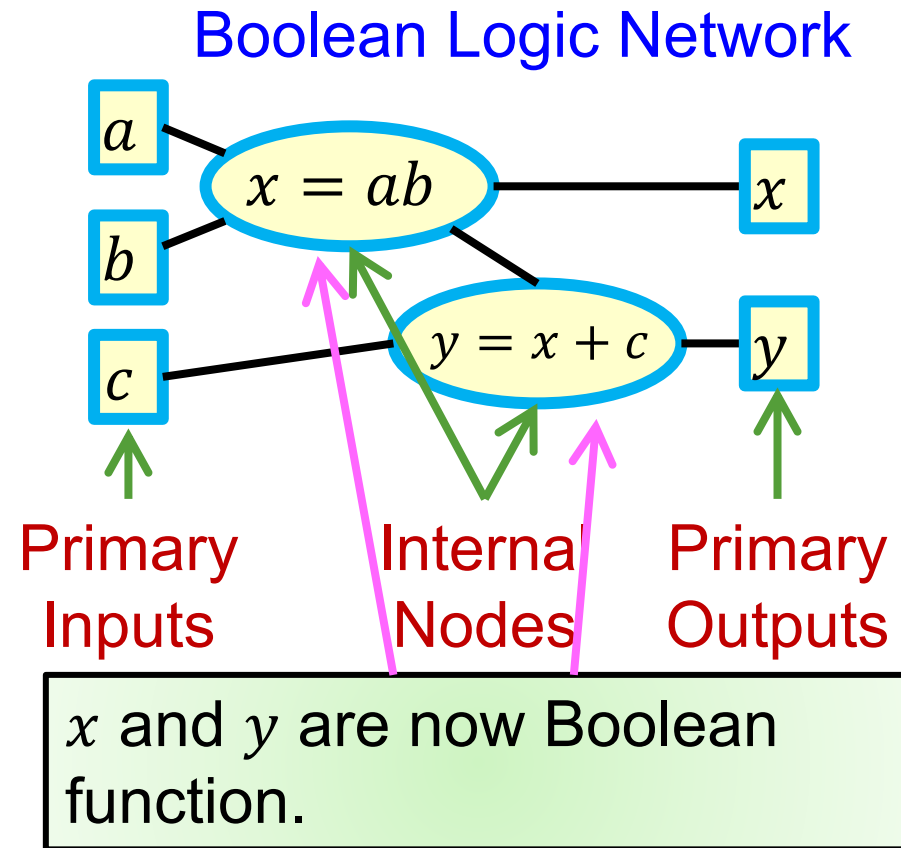
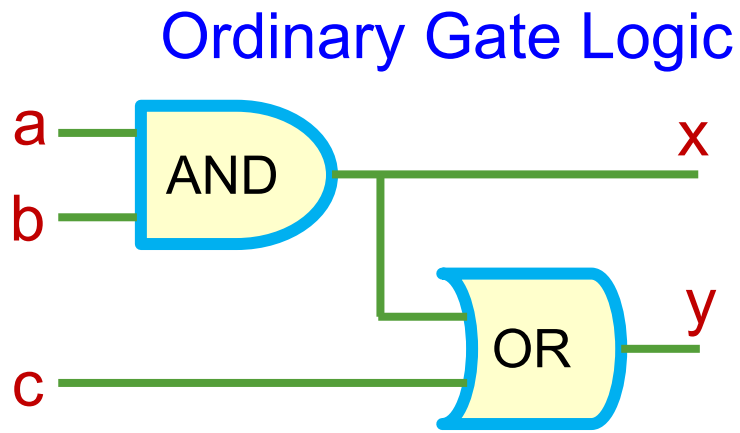


Outline

- Background on Logic Synthesis
- Technology-Independent Synthesis
- Technology Mapping
- Exact Synthesis

Boolean Logic Network Model

- A **graph** of connected blocks, like any logic diagram, but now individual component blocks are **2-level Boolean functions in SOP form**.

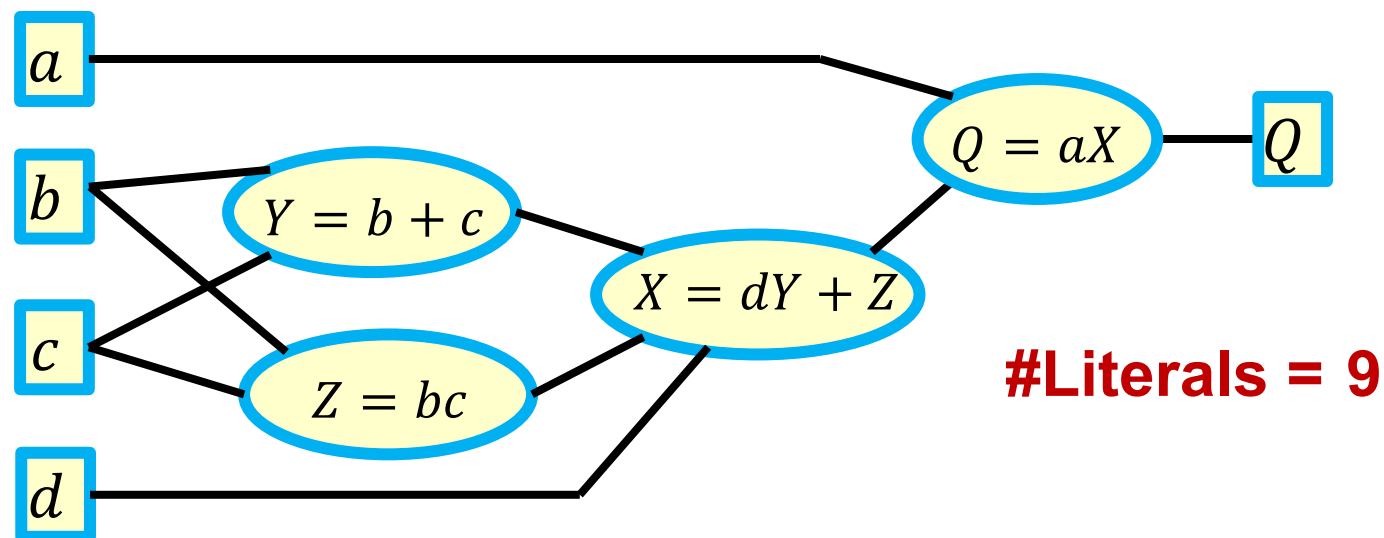


Boolean Logic Network: What to Optimize?

- ▶ A simplistic but surprisingly useful metric:

Total literal count

- Count **every** appearance of **every** variable on right hand side of “=” in **every internal node**.
- Delays also matter, but here only focus on **area**.



Optimizing Multilevel Logic: Big Ideas

- **Boolean logic network** is a **data structure**. So, what **operators** do we need?
- 3 basic kinds of operators:
 - **Simplify** network nodes: no change in # of nodes, just simplify insides, which are **SOP form**.
 - **Remove** network nodes: take “too small” nodes, **substitute them back** into fanouts.
 - This is not too hard. This is mostly manipulating the graph, simple SOP edits.
 - **Add** new network nodes: this is **factoring**. Take big nodes, split into smaller nodes.
 - This is a **big deal** and takes effort.

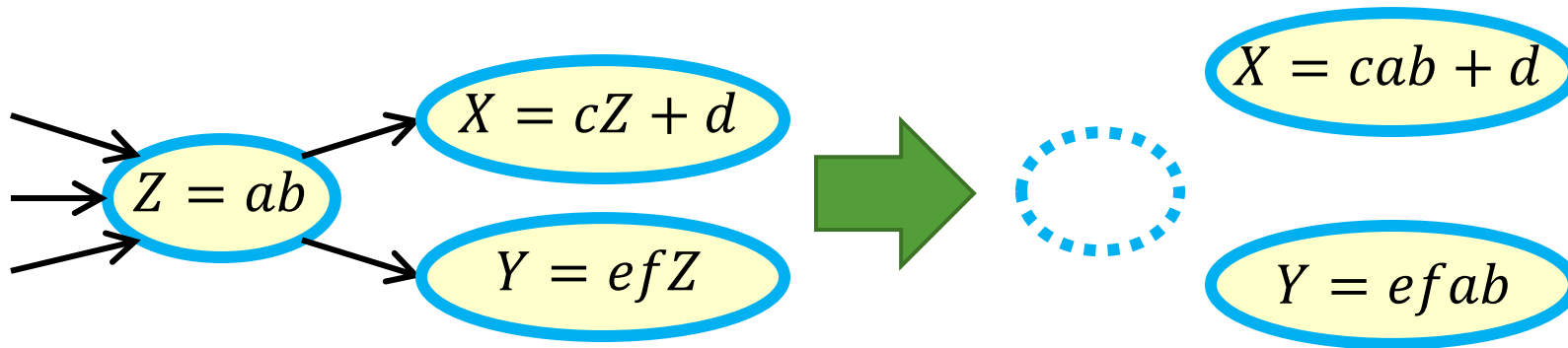
Simplifying a Node

- This is **2-level synthesis**.
- Just run ESPRESSO on 2-level form **inside** the node, to reduce # literals.
- As structural changes happen across network, "**insides**" of nodes may present opportunity to simplify.



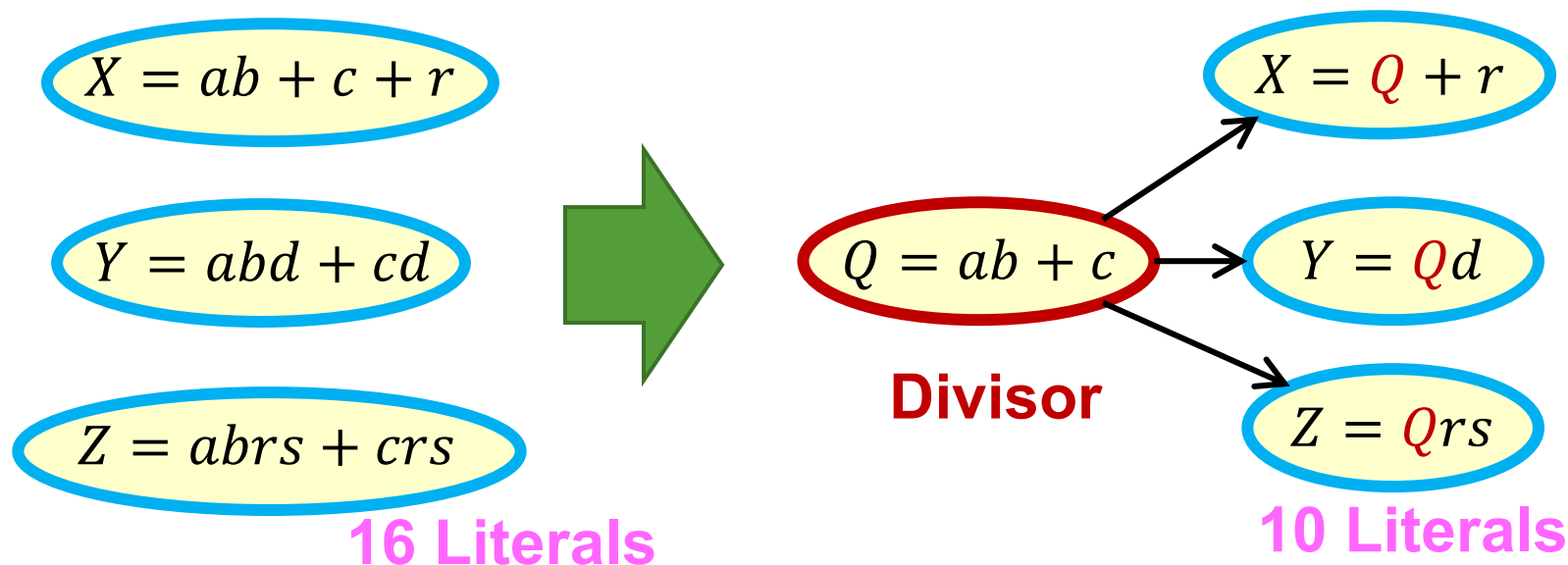
Removing a Node

- Typical case is you have a “**small**” factor which doesn’t seem to be worth making it a **separate** node.
- “**Push**” it back into its fanouts, make those nodes bigger, and hope you can use 2-level simplification on them.



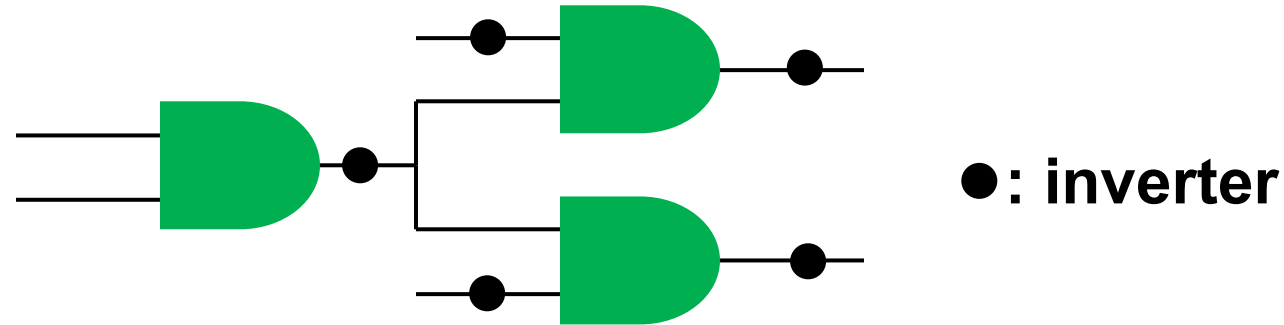
Adding new Nodes

- Look at existing nodes, identify **common divisors**, extract them, connect as fanins.
- Tradeoff: **more** delay (another level of logic), but **fewer literals** (less gate area).
- Based on **algebraic model**, **algebraic division**, **kernels**



AND-Inverter Graphs (AIGs)

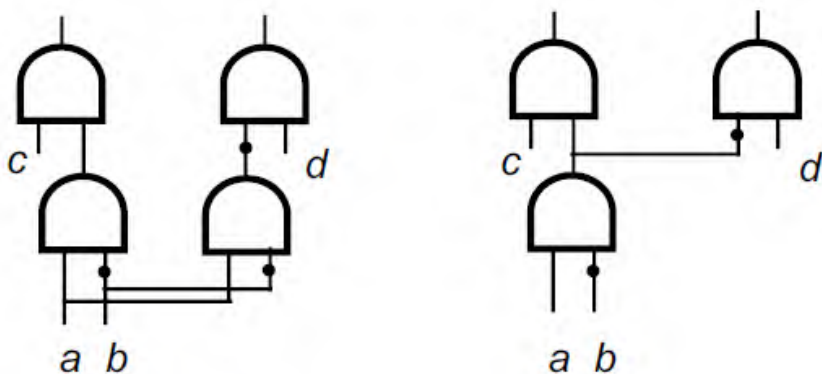
- Any Boolean function can be represented as an AIG!



- But AIG is not **canonical**
 - For the same function, many different AIG representations.
- For area reduction, minimize the **number of AND nodes**.
- Tool: ABC: A System for Sequential Synthesis and Verification
 - <http://www.eecs.berkeley.edu/~alanmi/abc>

AIG Structural Hashing (Strashing)

- When building AIGs, always add AND node
 - When an AIG is constructed without strashing, AND gates are added one at a time without checking whether AND gate with the same fanins already exists
- One-level strashing
 - When adding a new AND-node, check the hash table for a node with the same input pair (fanin)
 - If it exists, return it; otherwise, create a new node

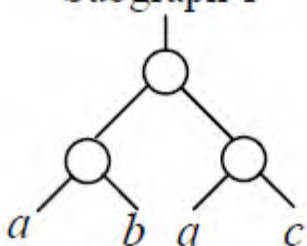
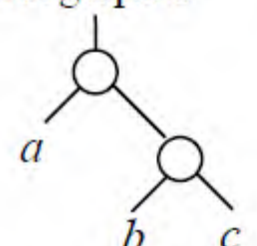
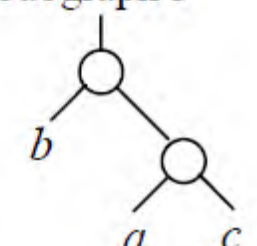


AIG Optimization Operations

- Rewriting ✓
- Resubstitution ✓
- Balancing
- Refactoring

Rewriting: Phase 1

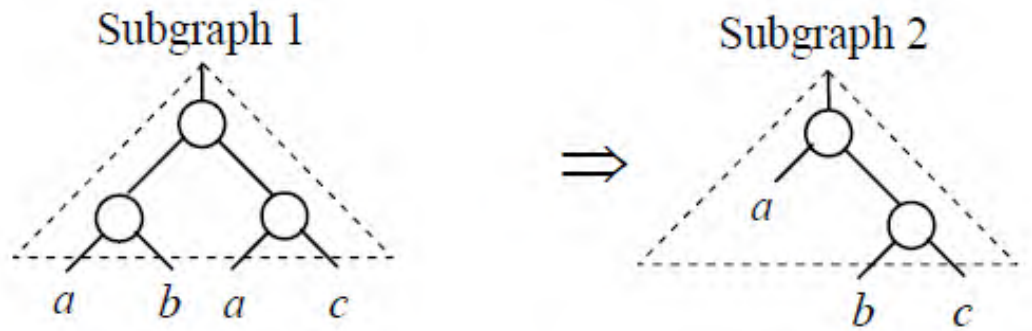
- All two-level AIG subgraphs are **pre-computed** and stored in a **hash table** by their functionality.
- The table contains all non-redundant AIG implementations of logic functions with four variables or less

Hash table	Function	All non-redundant AIG implementations		
		Subgraph 1	Subgraph 2	Subgraph 3
	abc			
	$a(b + c)$...		
		

Rewriting: Phase 2

- For each node in topological order, find its two-level AIG subgraph and compute its Boolean function.
- Use the function to access the hash table to find equivalent subgraphs.
- Try each subgraph, while taking into account logic sharing between the new subgraph nodes and the existing nodes.
- Choose the subgraph with largest save in # of nodes.

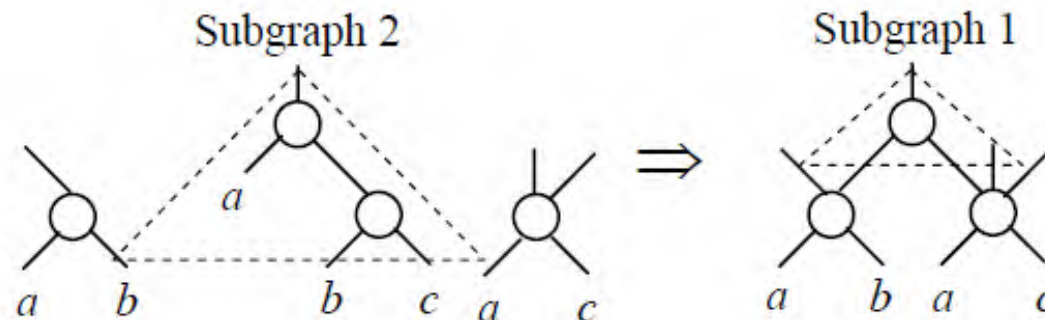
Example 1



Rewriting: Phase 2

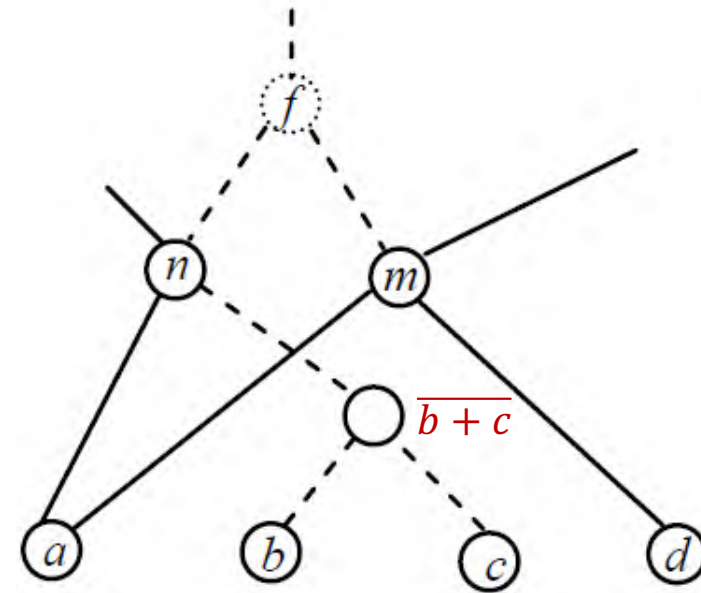
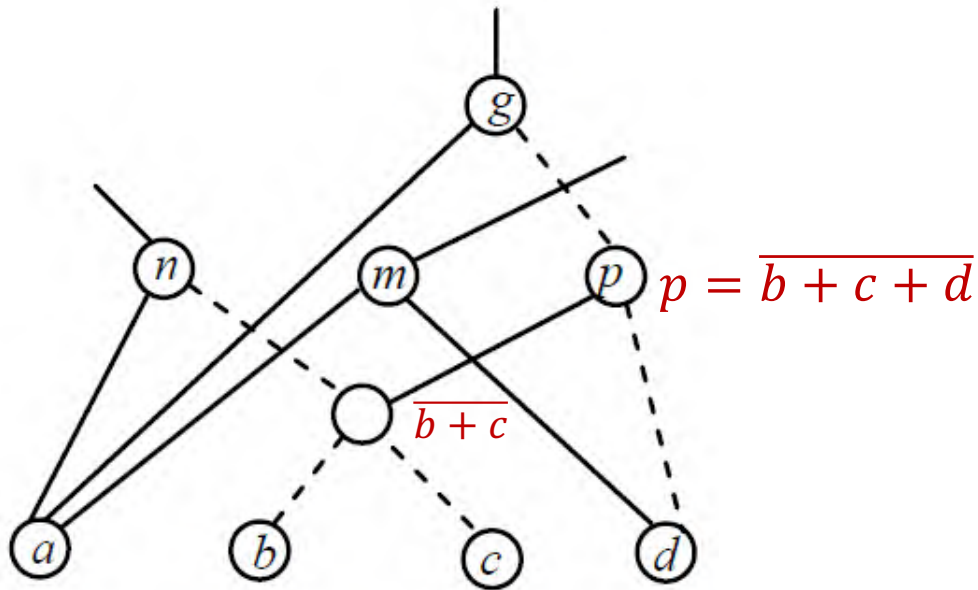
- For each node in topological order, find its two-level AIG subgraph and compute its Boolean function.
- Use the function to access the hash table to find equivalent subgraphs.
- Try each subgraph, while taking into account logic sharing between the new subgraph nodes and the existing nodes.
- Choose the subgraph with largest save in # of nodes.

Example 2



Resubstitution

- Express the function of a node using other nodes (divisors).
- Example:
 - Replace function $g = a(b + c + d)$ as $f = n + m$, where $n = a(b + c)$ and $m = ad$



More about AIG Optimization in ABC

- Some other operations, e.g., balancing and refactoring
- Write **scripts of basic operations**.
 - Do **several passes** of different optimizations over AIG.
 - E.g., balancing -> rewriting -> refactoring -> resubstitution
 - Lots of “art” in the engineering of these scripts.
- To explore more:
 - Visit ABC webpage: <http://www.eecs.berkeley.edu/~alanmi/abc>

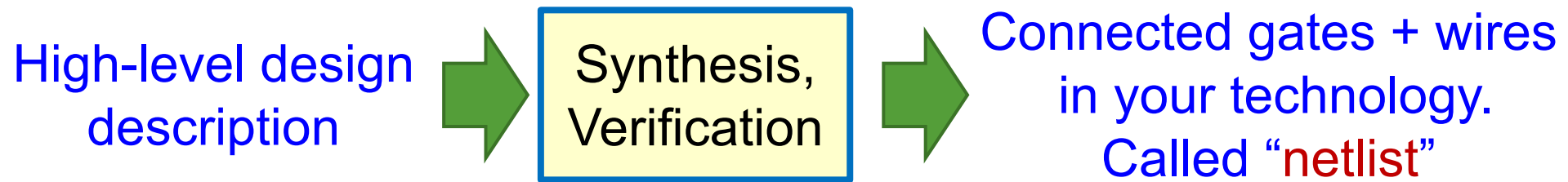
Outline

- Background on Logic Synthesis
- Technology-Independent Synthesis
- **Technology Mapping**
- Exact Synthesis

From Logic... To Layout...

► What you know...

- Computational Boolean algebra, representation, some verification, some synthesis.
- This is what happens in the “**front end**” of the ASIC design process.

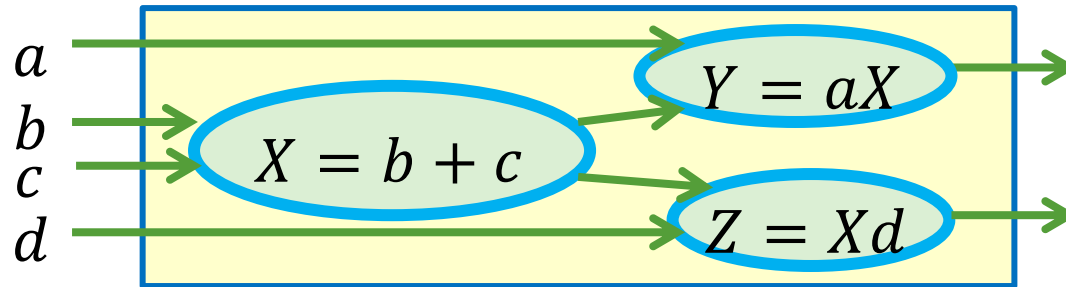


► One key **connecting** step:

- How to **transform** result of multi-level synthesis into real gates for layout task.
- Called: **Technology Mapping**, or **Tech Mapping**.

Tech Mapping: The Problem

- Multi-level model is still a little **abstract**.
 - Structure of the Boolean logic network is fixed.
 - ESPRESSO-style 2-level simplification done on each node of network.
 - ... but that still does **not** give us the actual **gate-level netlist**.

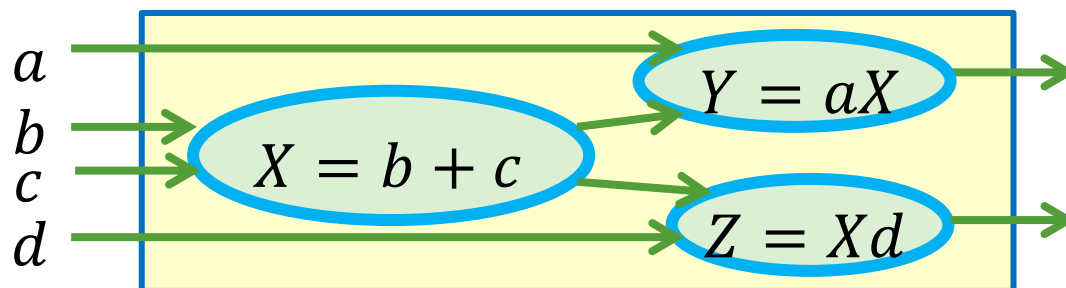


Tech Mapping: Problem

- Suppose we have these gates in our “**library**”.
 - This is called “**the technology**” we are allowed to use to build this optimized netlist.



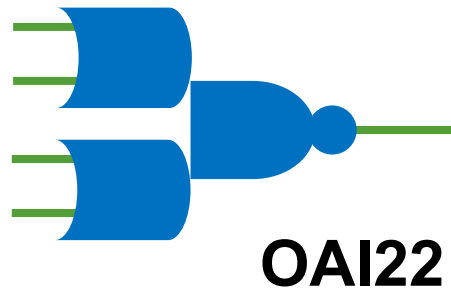
- Note: OA21 is an OR-AND, a so-called **complex gate** in our library.



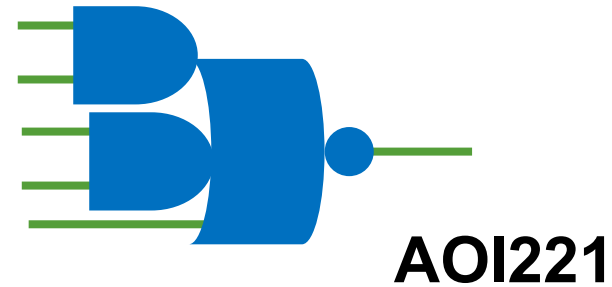
How do we build the 2 functions specified in this Boolean Logic Network using **only** these gates from our library?

Aside: Complex Gates

OR-AND-Inverter

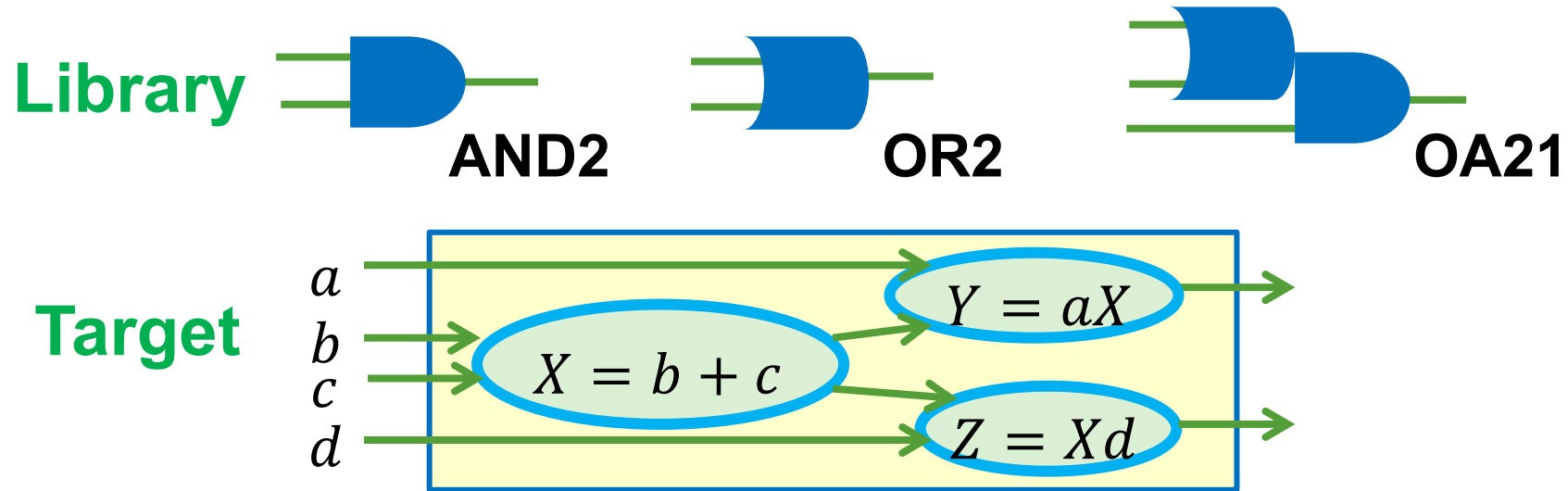


AND-OR-Inverter

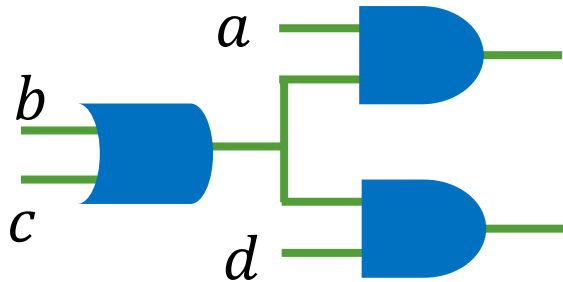


- In CMOS, OAI and AOI gate structures are **efficient** at transistor level.

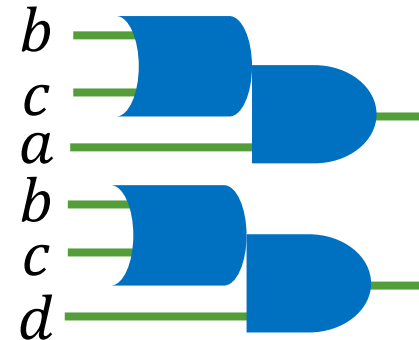
Tech Mapping: Simple Example



Obvious Mapping



Non-obvious Mapping



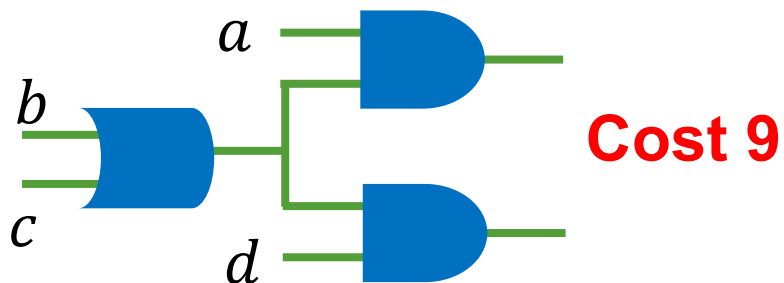
Tech Mapping

► Why choose a **non-obvious** mapping?

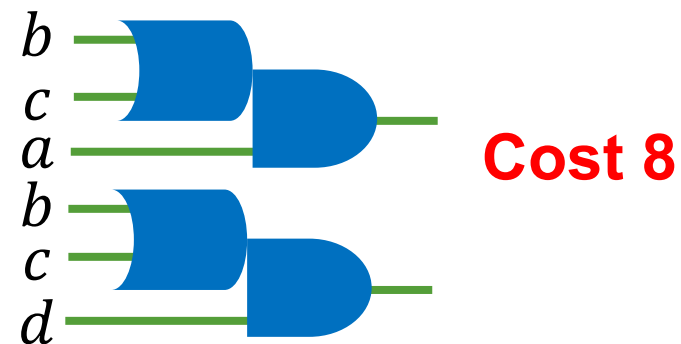
- Answer 1: **Cost**. Suppose each gate in library has a cost associated with it, e.g., the **silicon area** of the standard cell gate.



Obvious Mapping

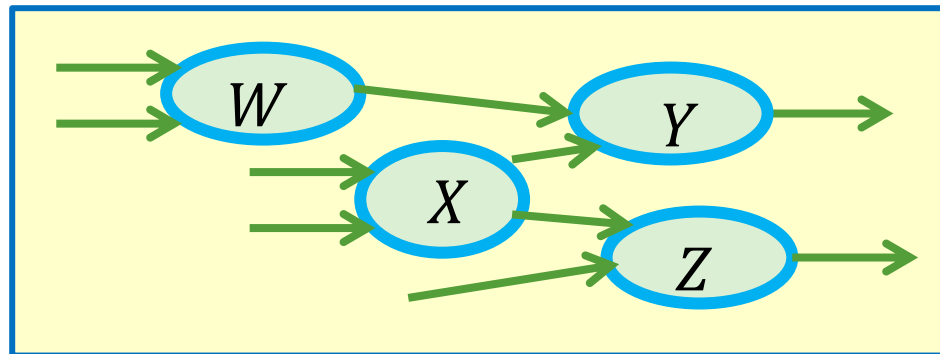


Non-obvious Mapping



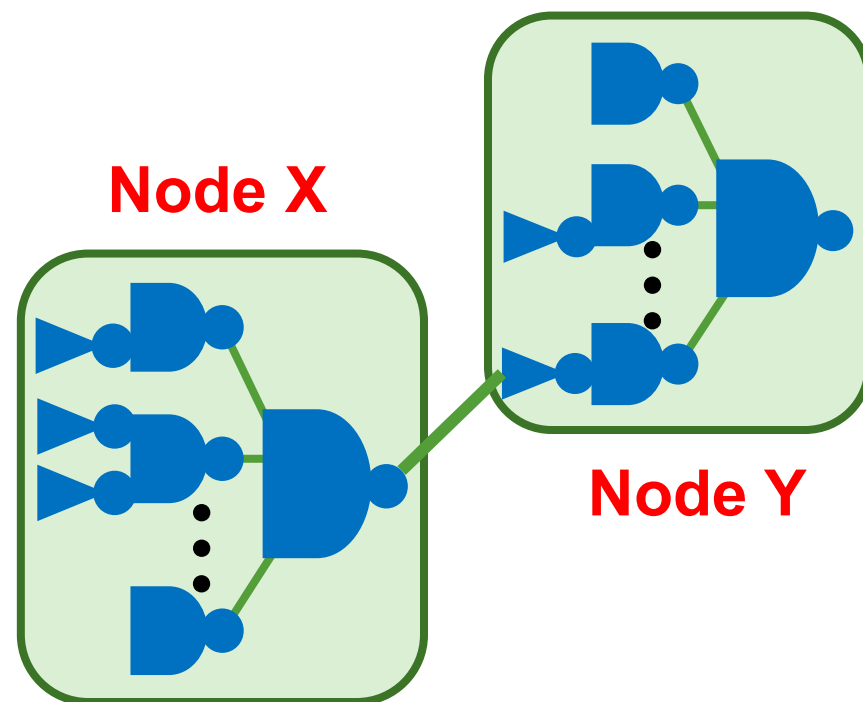
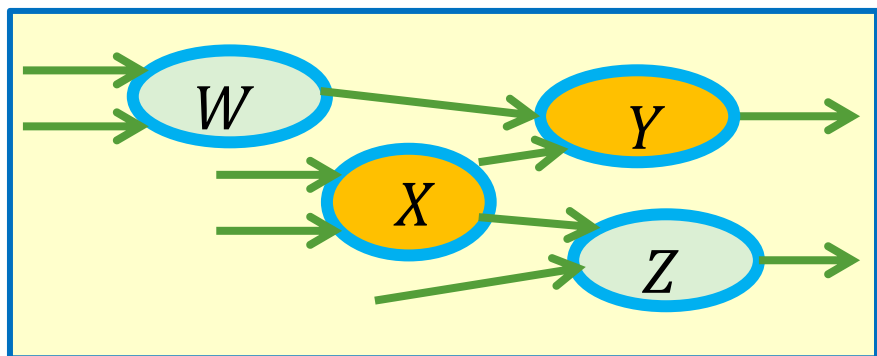
What “Multilevel Synthesis” Does?

- ▶ Helpful “mental” model to use: Multi-level synthesis does this...
 - **Structures** the multiple-node Boolean logic network “**well**”.
 - **Minimizes** SOP contents of each vertex in the network “**well**”, ie, # of literals.
 - **But** it does **not create real logic gates**.
 - This result is called: “**uncommitted**” logic, or “**technology independent**” logic.

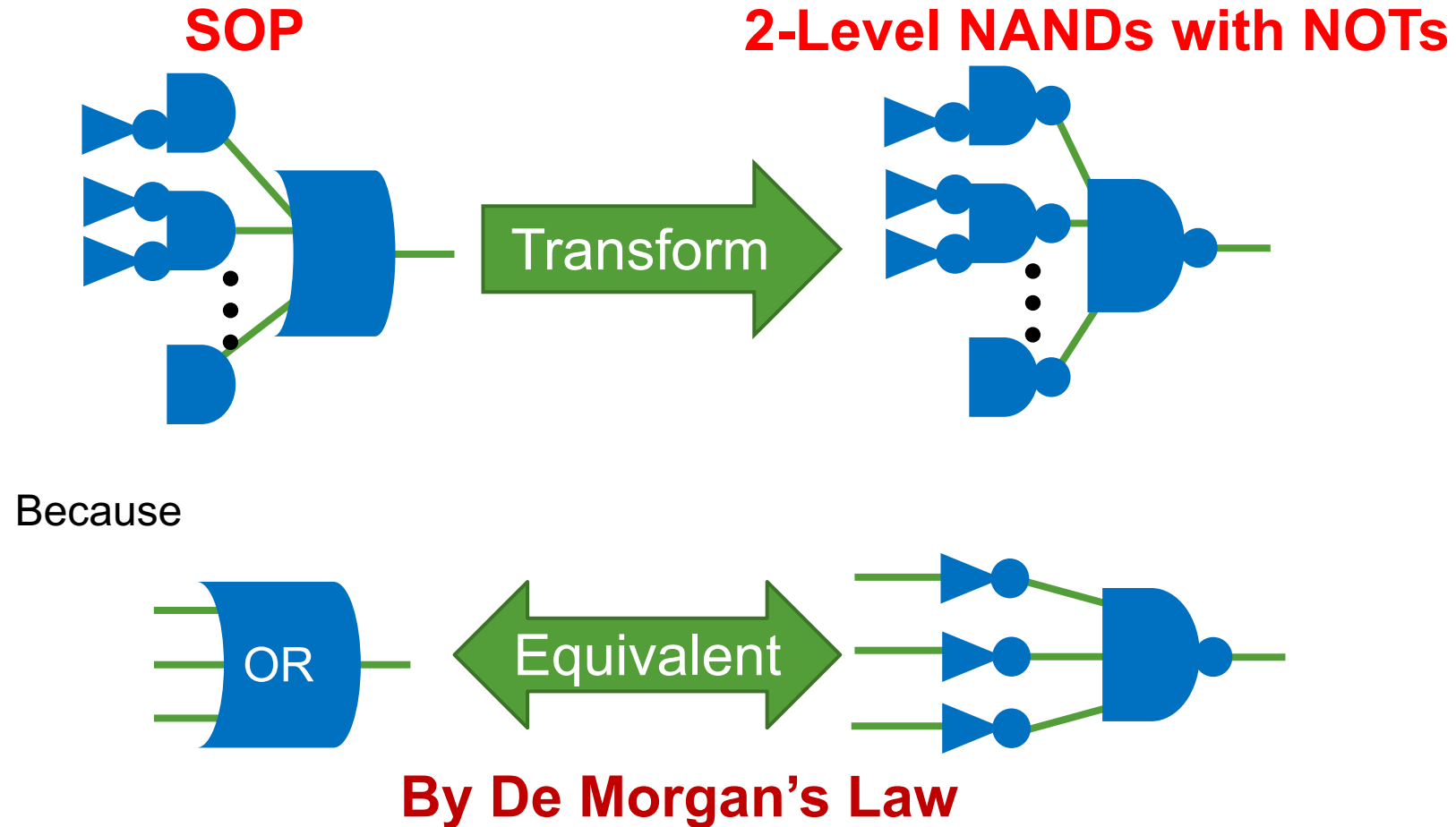


What “Technology Mapping” Does?

- First, we transform **uncommitted logic** into simple, **real** gates.
 - We transform **every** SOP form in **each** node into **NAND** & **NOT** gates. Nothing else!

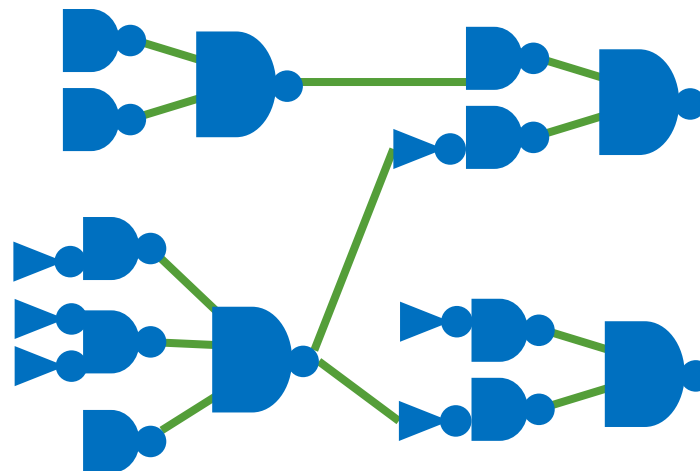
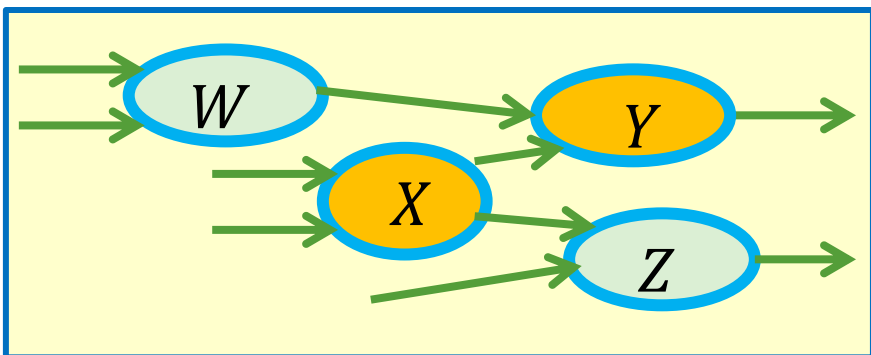


Aside: We Can Map SOP into NANDs & NOTs

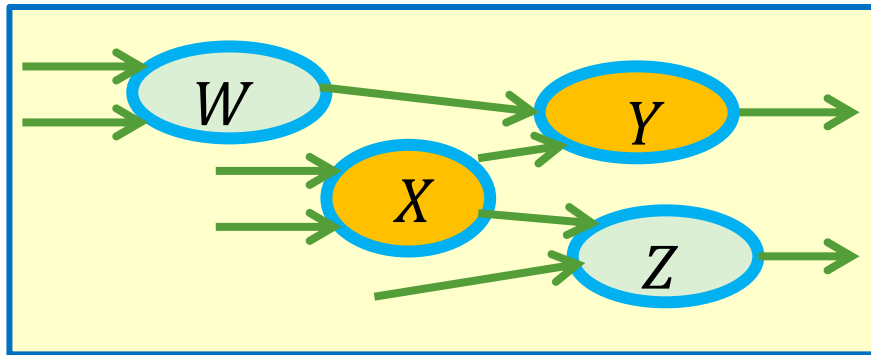


What “Technology Mapping” Does?

- By transforming every SOP form in each node into NAND & NOT gates ...
 - ... Boolean logic network **disappears**. W, X, Y, Z boundaries go away.
 - We have one BIG “**flat**” network of **NANDs** and **NOTs**. This is what we are going to map.



-
- A complex logic circuit diagram featuring several blue AND gates of various sizes connected by green lines. The circuit includes a large central AND gate with four inputs, and several smaller AND gates arranged around it, some with two inputs and others with three. The connections form a network that likely implements a specific logical function.

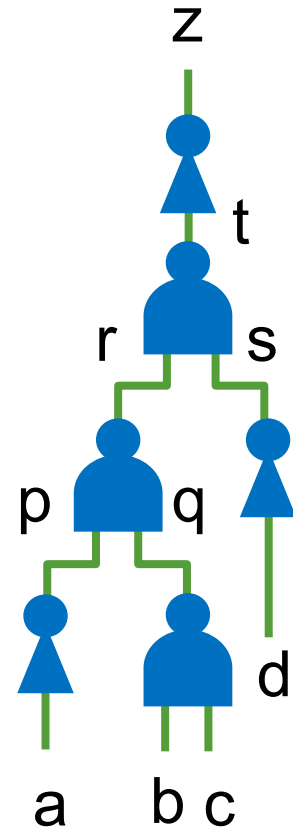


Technology Mapping as Tree Covering

- One famous, simple model of problem:
 - Your logic network to be mapped is **a tree of simple gates**.
 - We assume uncommitted form is **2-input NAND** (“**NAND2**”) and **NOT** gates, only.
 - Your library of available “real” gate types is also represented in this form.
 - Each gate is represented as a **tree** of **NAND2** and **NOT** gates, with associated **cost**.
- Method is surprisingly **simple** and **optimal**.
 - Reference: Kurt Keutzer, “DAGON: Technology Binding and Local Optimization by DAG Matching,” Proc. ACM/IEEE Design Automation Conference (DAC), 1987.

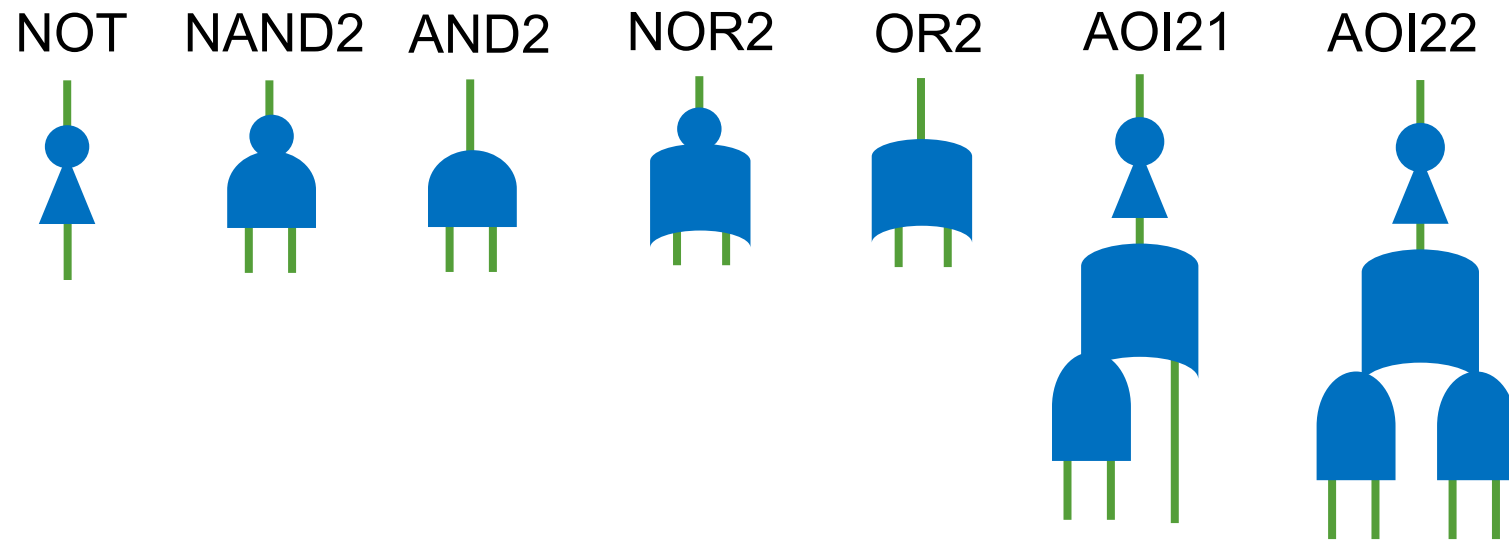
Tree Covering Example: Start with...

- Here is your uncommitted logic to be **tech mapped**.
 - This is what results from our multi-level synthesis optimization...
 - ... after replacing all SOP forms in the network nodes with **NAND2/NOT**.
 - Called the **subject tree**.
 - (Restrict to **NAND2** to keep this simple. Also label not only inputs but all internal wires too.)



Tree Covering: Your Technology Library

- And, here is a very simple **technology library**.

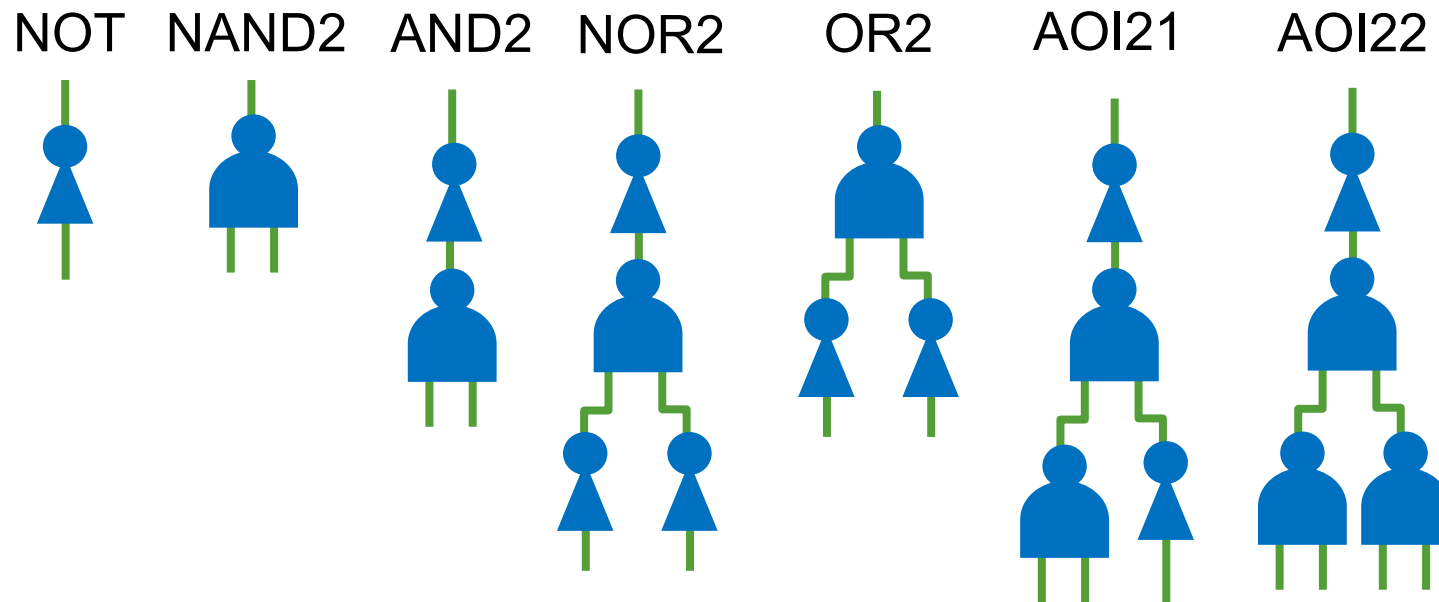


- First problem: this is **not** in the required **NAND2/NOT-only** form. Must transform.

Tree Covering: Representing Library

► Transforming to NAND2/NOT form is **easy**.

— Just apply **De Morgan's law**.



► Each library element in this form is called a **pattern tree**.

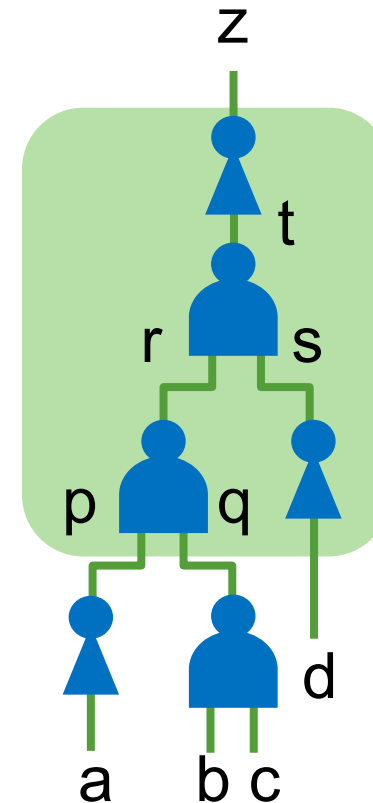
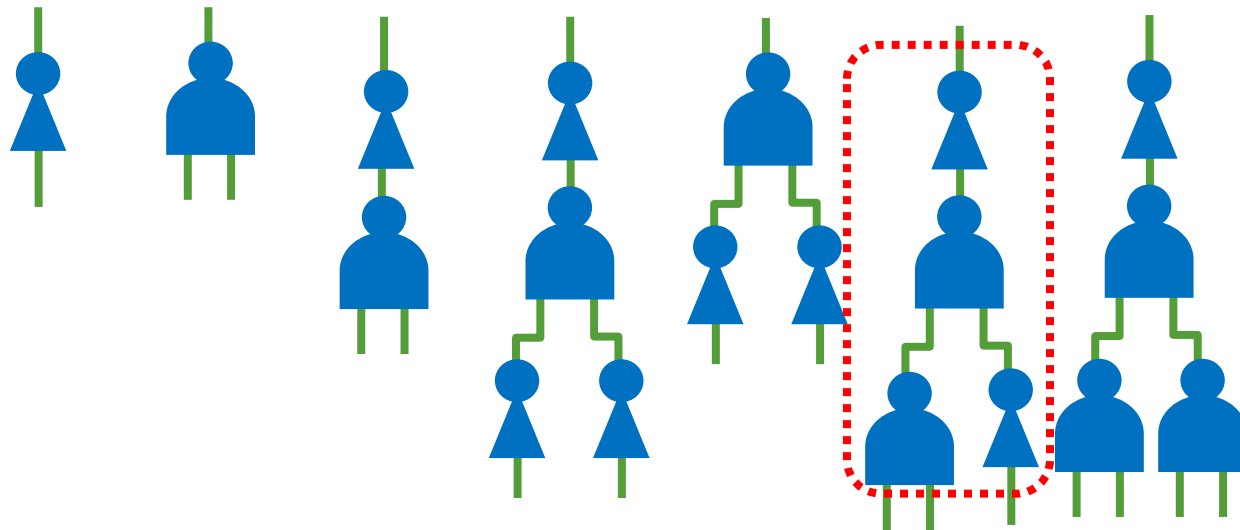
Essential Idea in “Tree Covering”

➤ **Avoid** any **Boolean algebra**!

➤ Just do “**pattern matching**”.

- Find where, in subject graph, the library pattern “matches”.
- NAND matches NAND, NOT matches NOT, etc.
- This is called: **structural mapping**.

NOT NAND2 AND2 NOR2 OR2 AOI21 AOI22

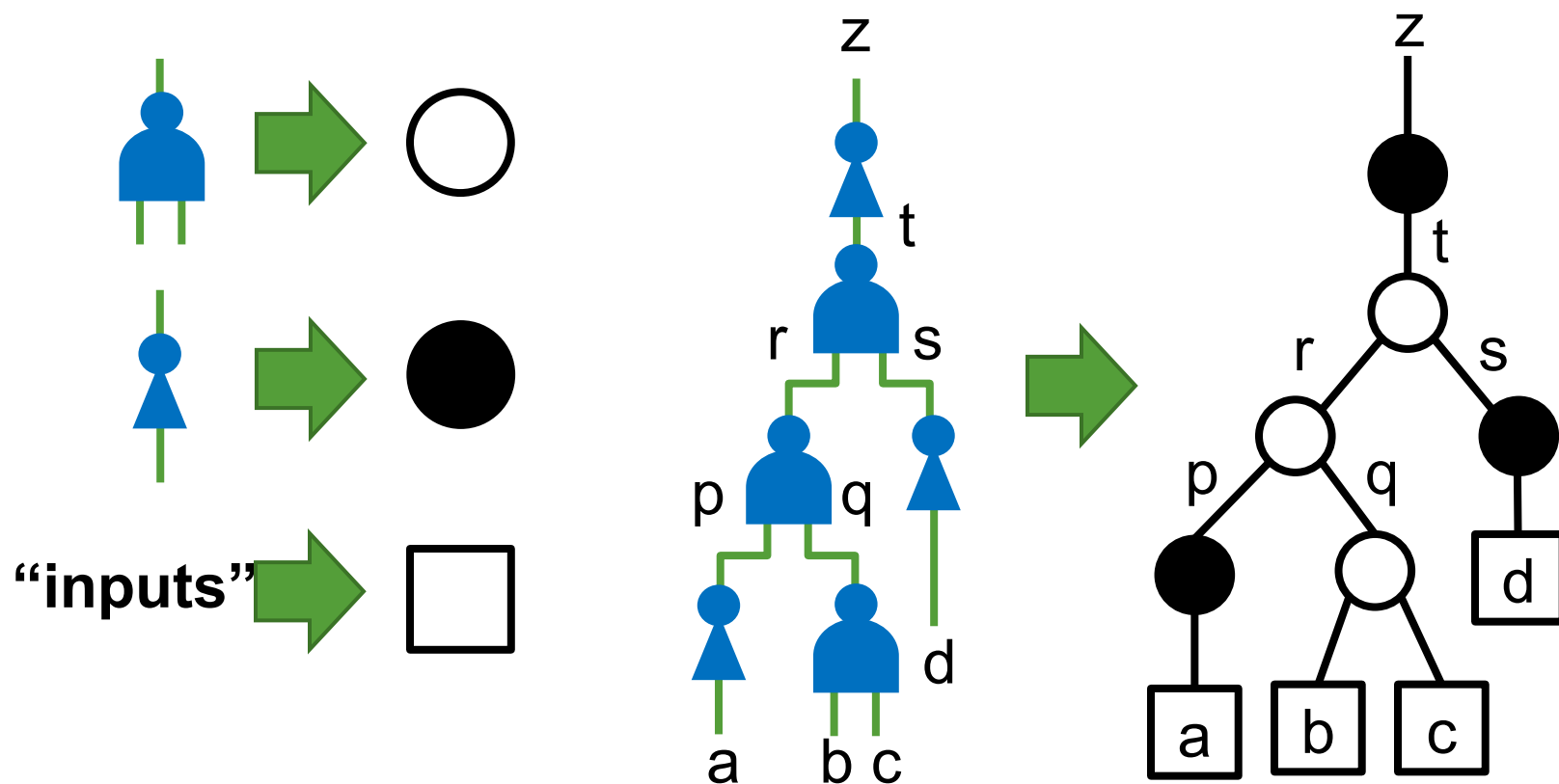


The General “Tree Covering”

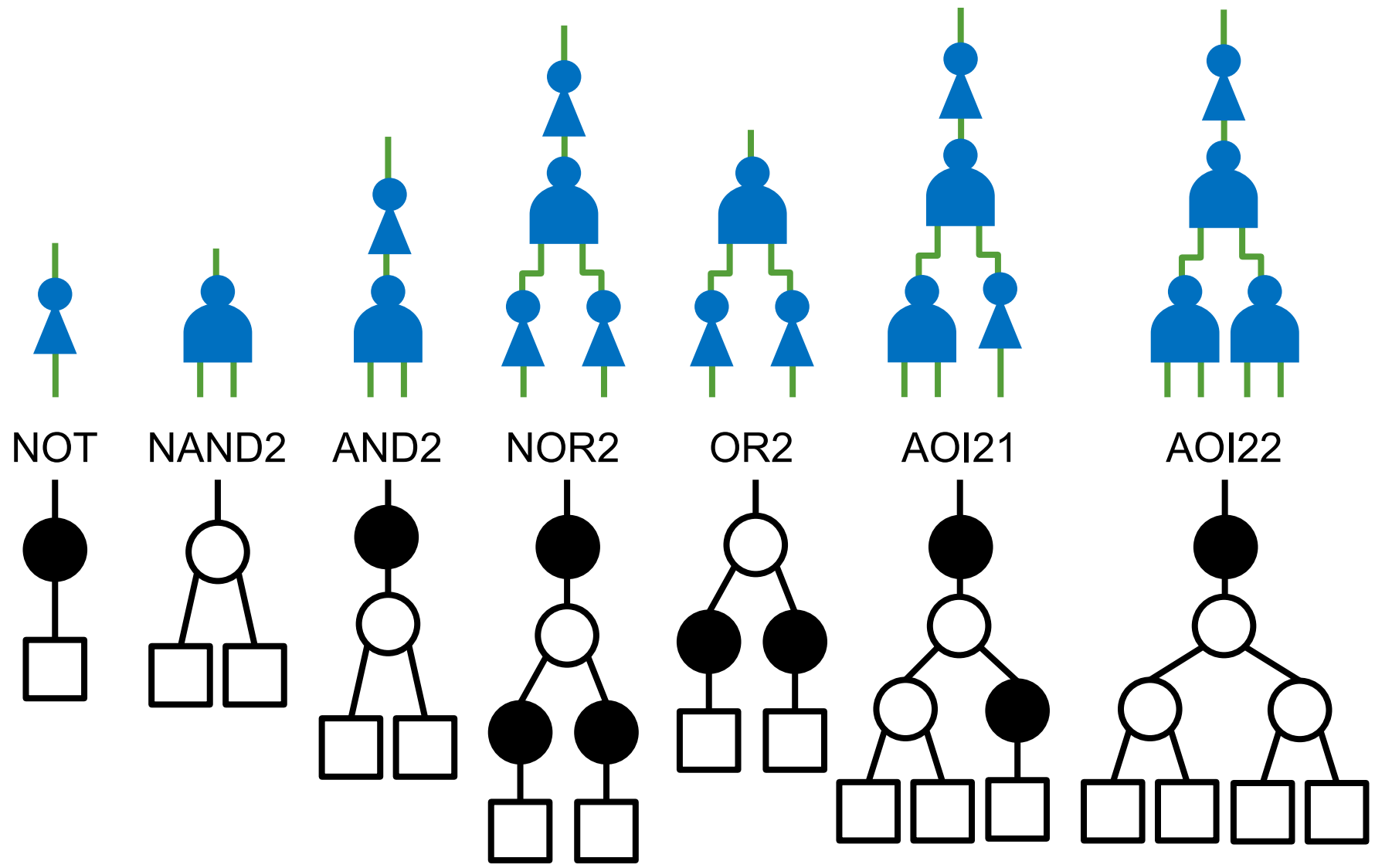
- This is called a **structural tech mapper**.
- Why?
 - Because there is no Boolean algebra here!
 - We just match the gates, wires in a simple **pattern-match** way.
- Result
 - Surprisingly simple covering algorithm for **cost-optimal** cover.
- ... But first, lets simplify the way we draw these, to emphasize “structural” match.

Representing Trees for Covering

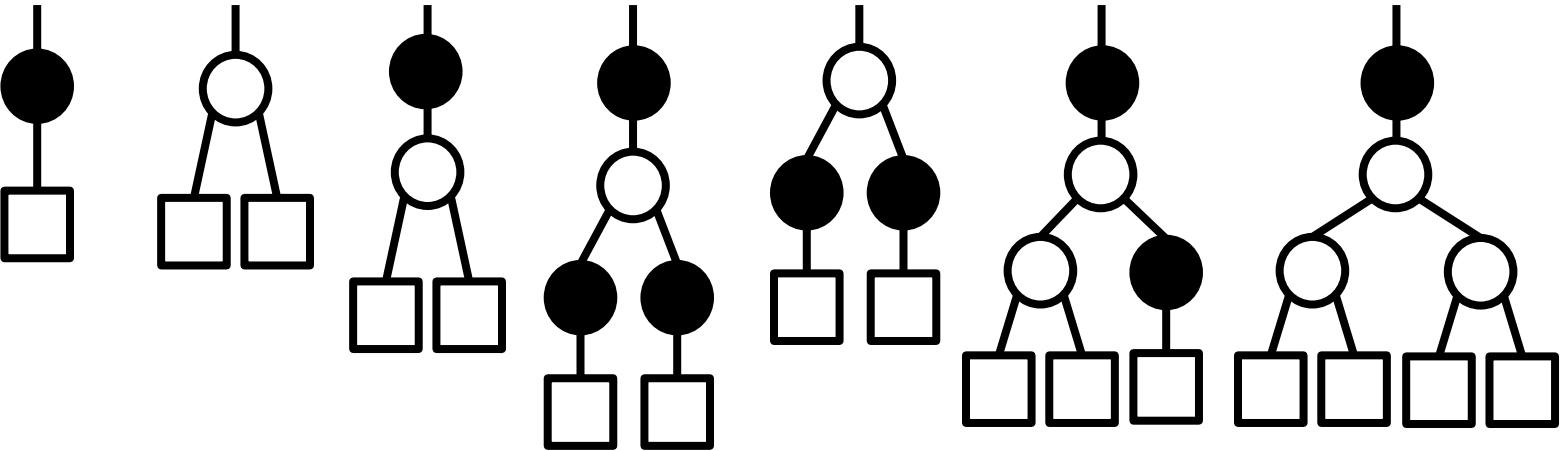
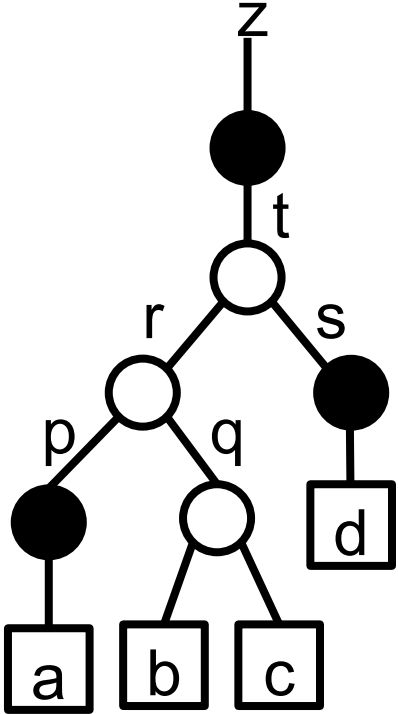
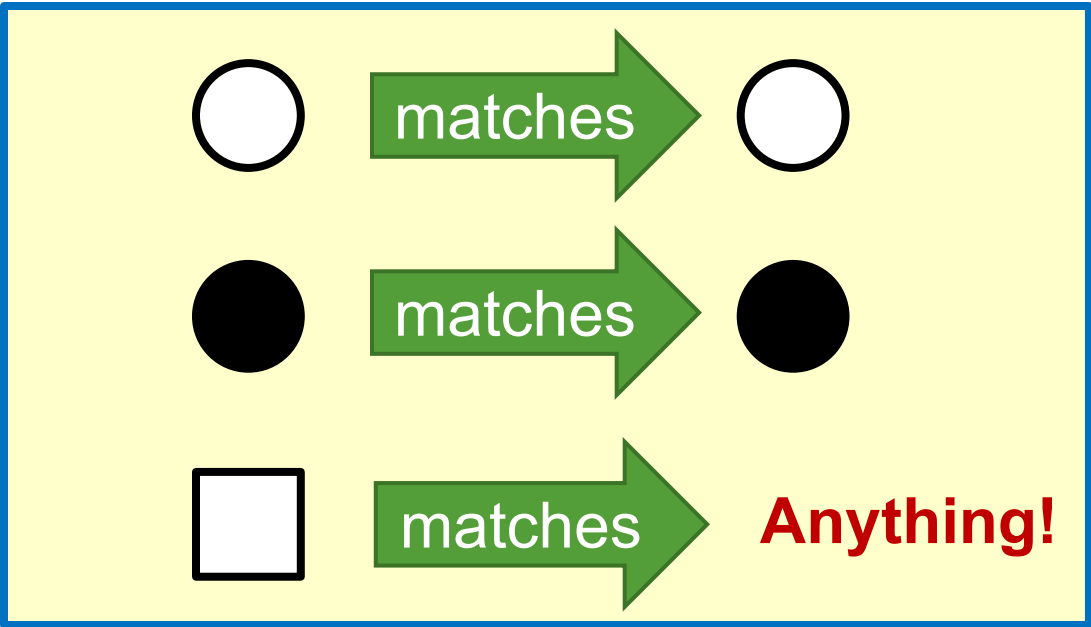
- Only **three** kinds of structures that we need **match** in any tree.



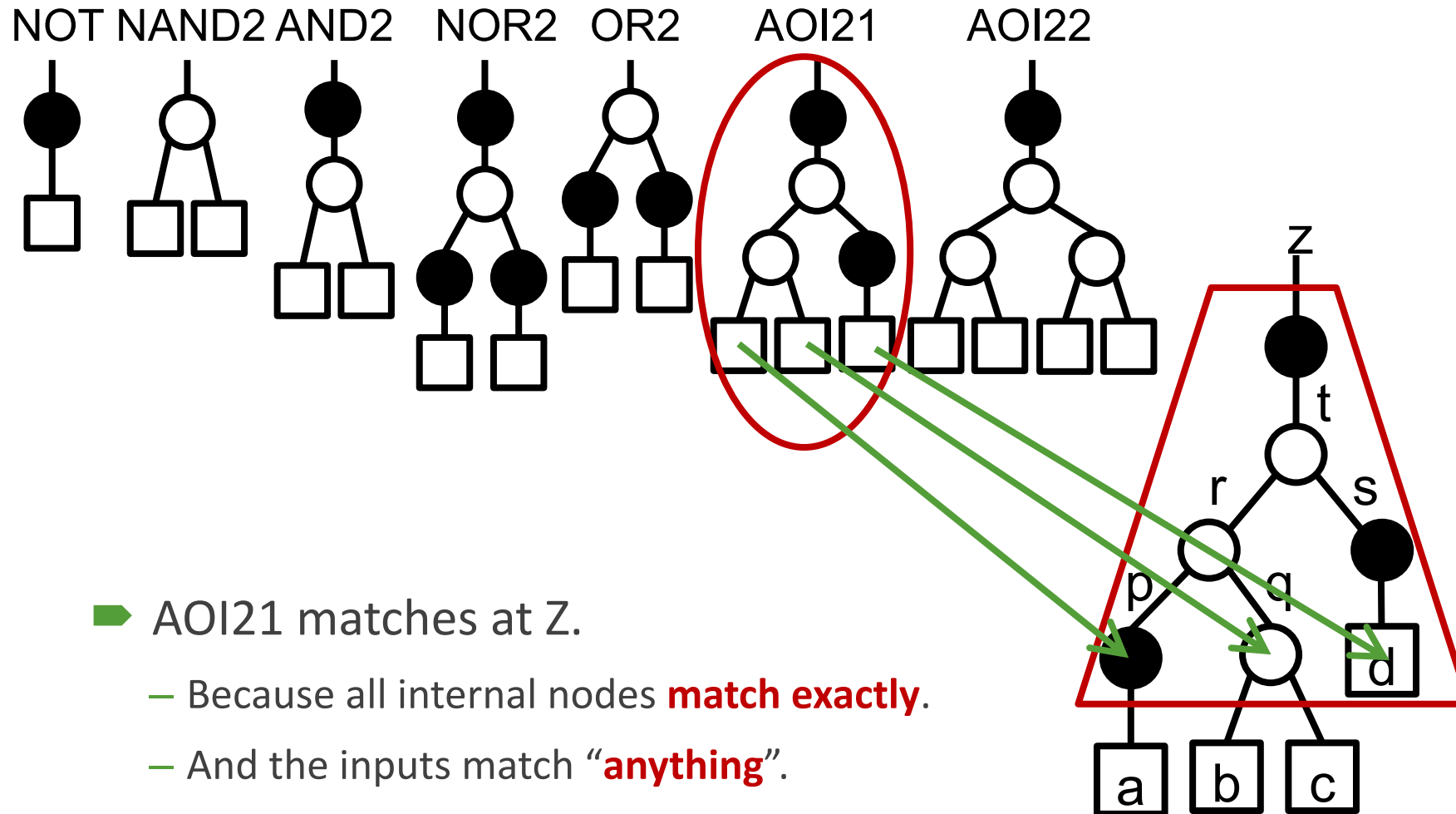
Represent Library in this Same Style



Structural Mapping Rules

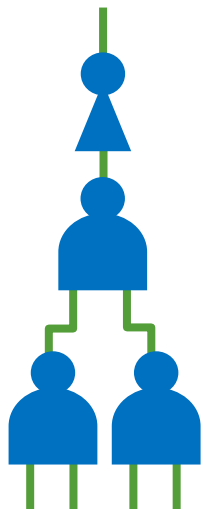


How a “Target Gate” Matches Subject Tree



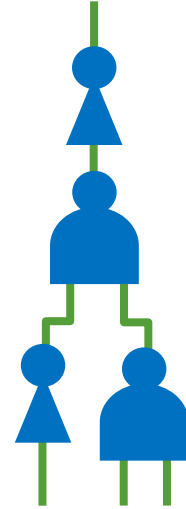
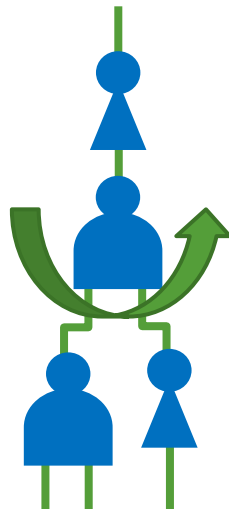
Be Careful: Symmetries Matter!

AOI22

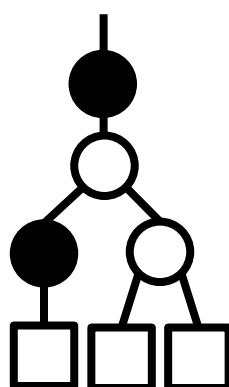
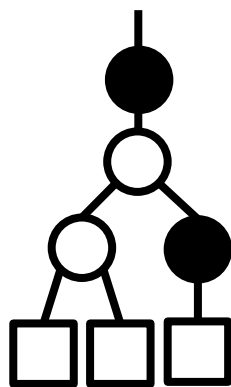
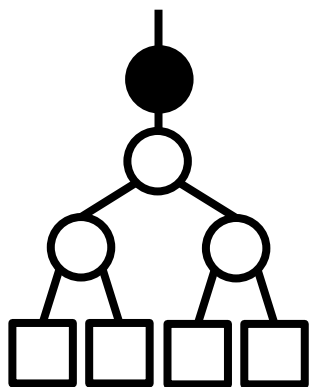


Symmetric
tree,
1 way to match

AOI21



Not
symmetric
2 ways to
match

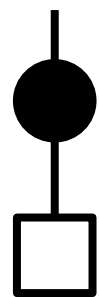


Rules for a Complete Tree Cover

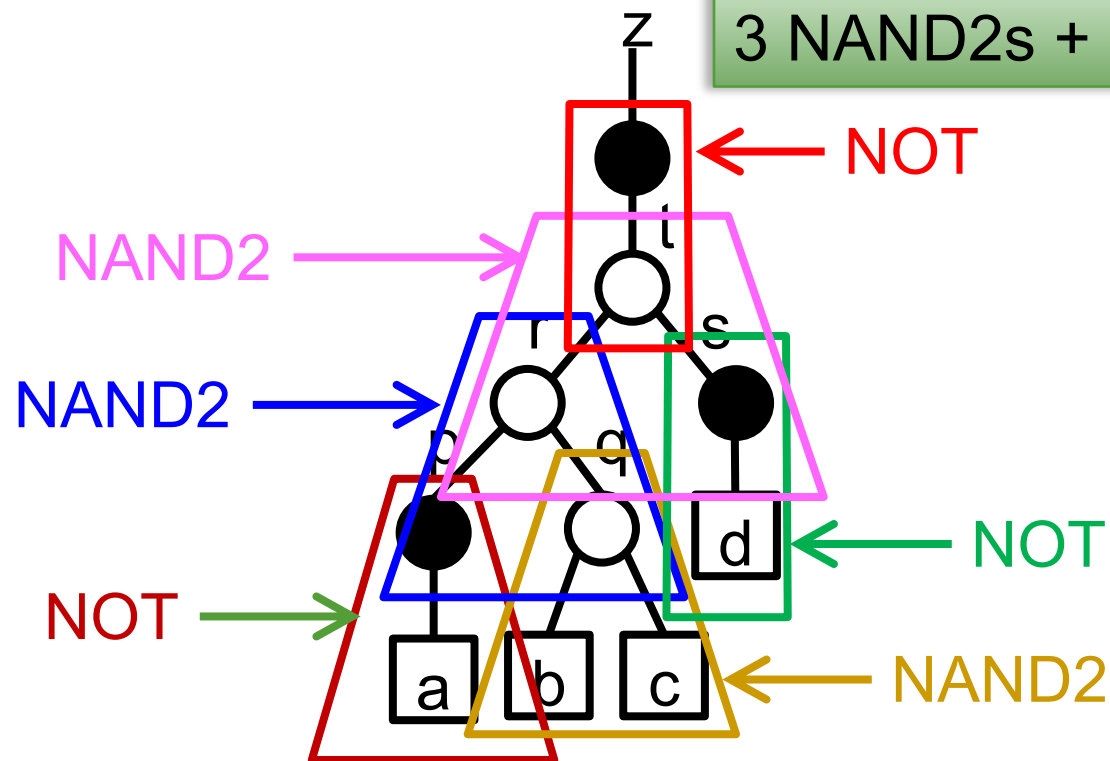
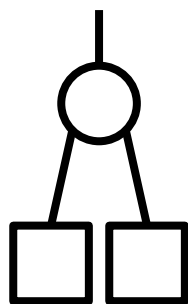
- Every node in subject tree is **covered** by some library tree.
- **Output** of every library gate **overlaps input** of next library pattern.

One correct tree cover:
3 NAND2s + 3 NOTs

NOT



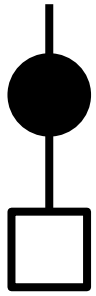
NAND2



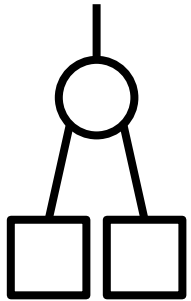
Rules for a Complete Tree Cover

- Note: usually there are **many different** legal covers.
- Which one do we choose? The one **with minimum cost**.

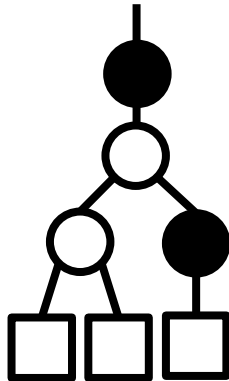
NOT



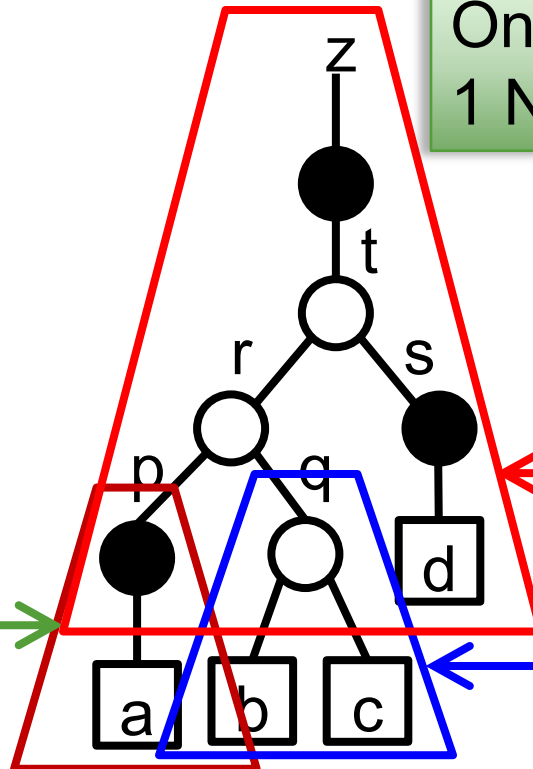
NAND2



AOI21



NOT



One **different** tree cover:
1 NAND2 + 1 NOT + 1 AOI21

AOI21

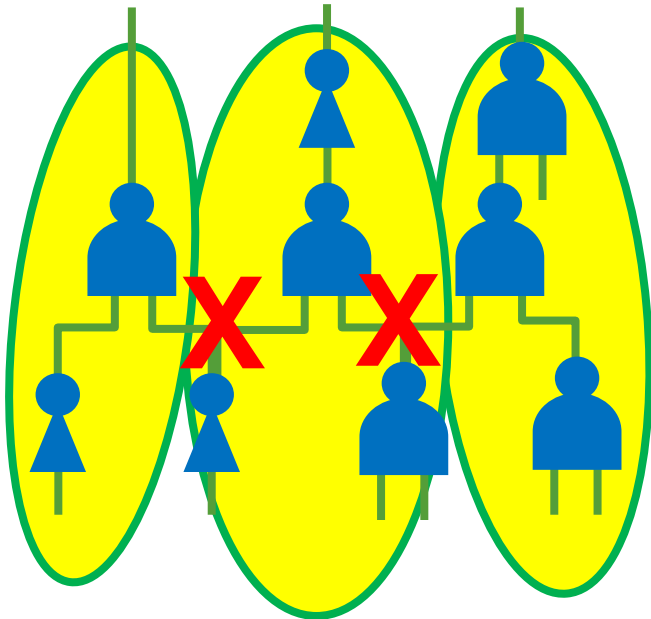
NAND2

Tech Mapping via Tree Covering

- What do we need for a **complete** algorithm?
- **Treeifying** the input netlist
- **Tree matching**
 - For each node in the subject tree, find pattern trees in library that **match**.
- **Minimum-cost covering**
 - Assume you know what can match at each node of subject tree
 - ... so, which ones do you pick for a **minimum cost** cover?

Treeifying the Netlist

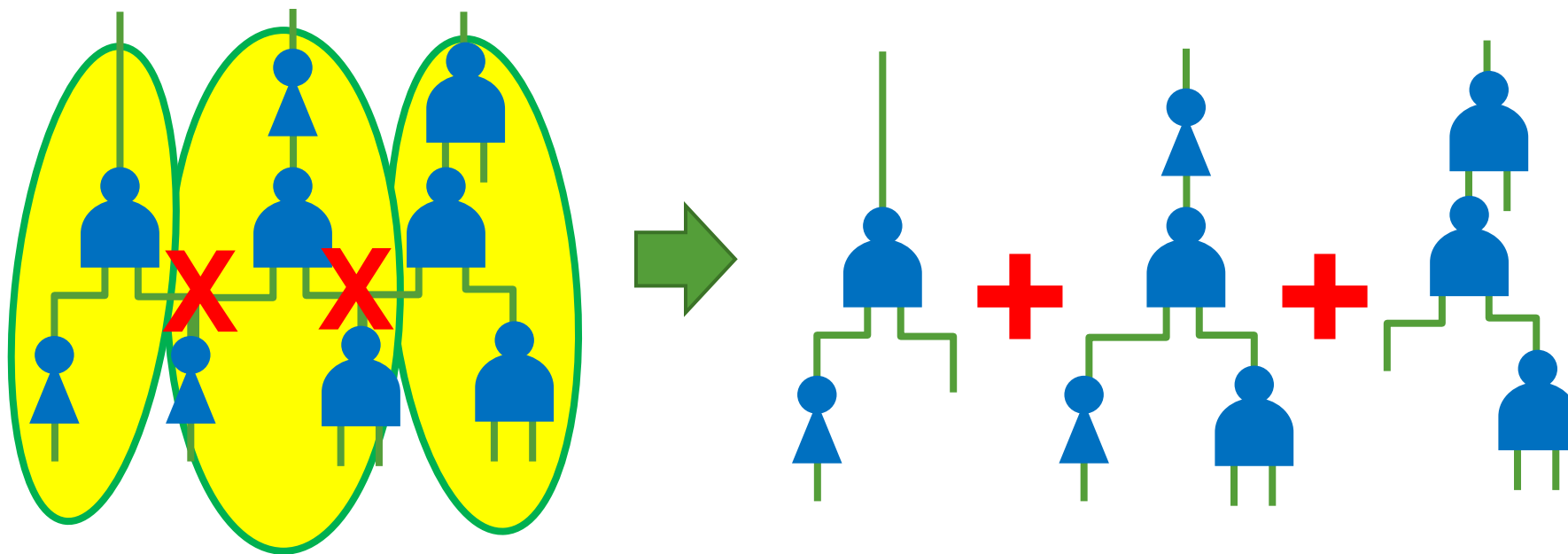
- These algorithms **only** work on trees, not on general graphs.
 - Note: general gate netlists are **Directed Acyclic Graphs (DAGs)**.
- **Treeifying**: every place you see a gate with fanout > 1, you **need to split**.



Must **split** this DAG into **3 separate trees**, map each separately.

This entails some clear loss of optimality, since **cannot** map across trees.

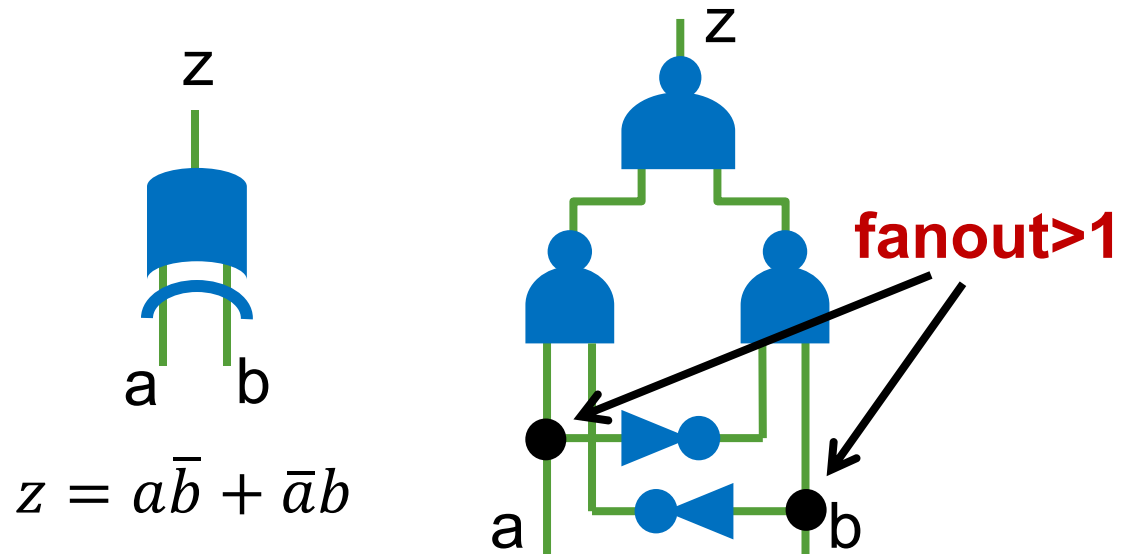
Treeifying Netlist: Result



- We're going to map these **3 trees** separately.
- Loses some optimality.
 - There are ways around these, but we won't discuss these.

Aside: How Restrictive is “Tree” Assumption?

- Subject graph and each pattern graph **must be trees**.
 - Subject tree must be treeified.
- What about **pattern trees**?
 - Are there common, useful gates that **cannot** be trees?
 - Yes! For example, XOR gate.
 - There are tricks to deal with this, but for us, these are forbidden!



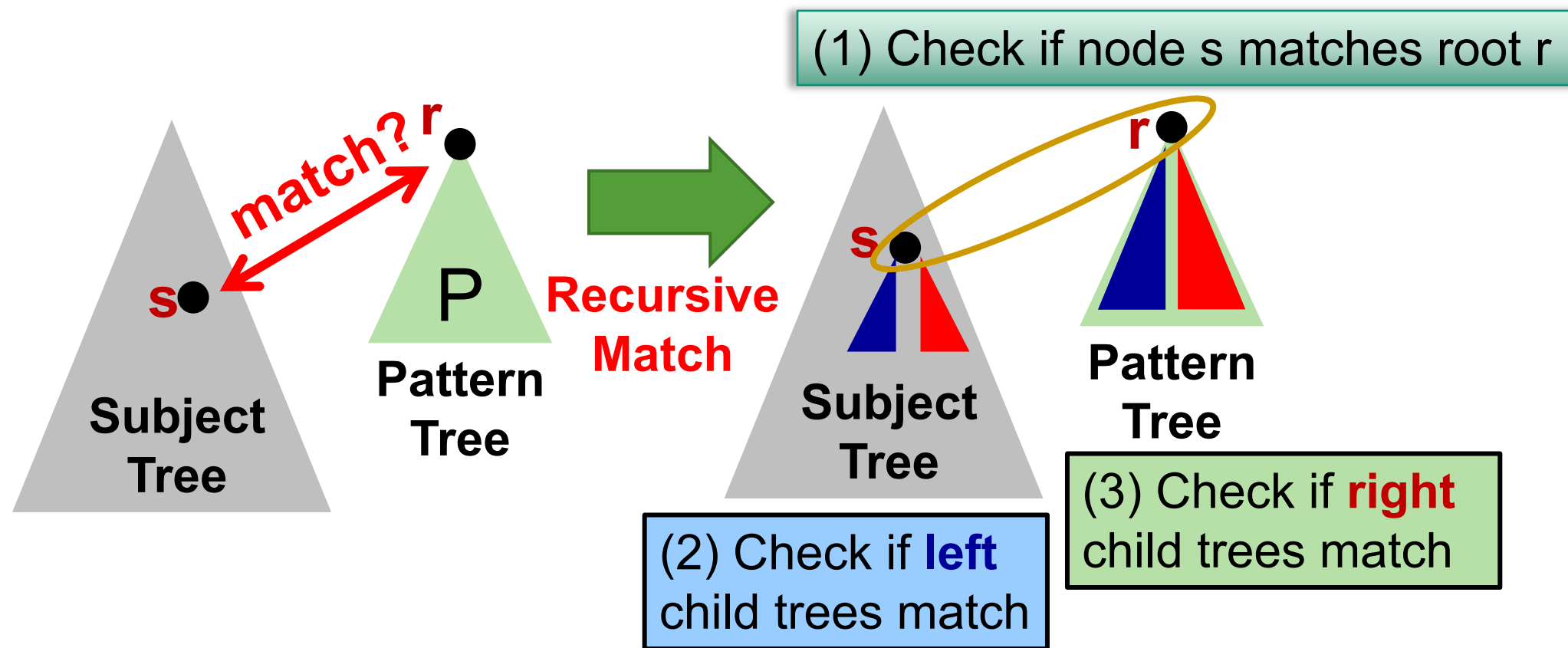
So, no XOR gates for mapping!

Tree Matching

- **Goal:** Determine, for every node in subject tree, what library gate can **match** (structurally).
- Straightforward approach: **Recursive matching**
 - Simple idea is to just try **every library gate** at **every node of subject tree**.
 - Library gates are small patterns – this is not too much work.
 - **Recursive** means: match **root** of subject with **root** of pattern, and then **recursively match children** of subject to **children** of pattern.

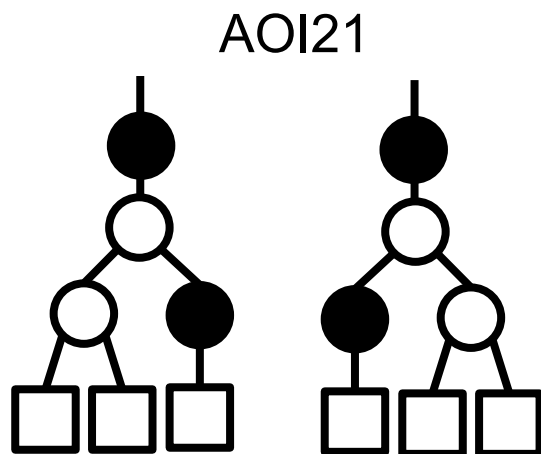
Recursive Tree Matching

- Does library pattern tree P match node s in our subject tree?
 - First, check if node s matches root r of pattern P .
 - If so, **recursively** match **left child trees** of s and r , and then **right child trees** of s and r .



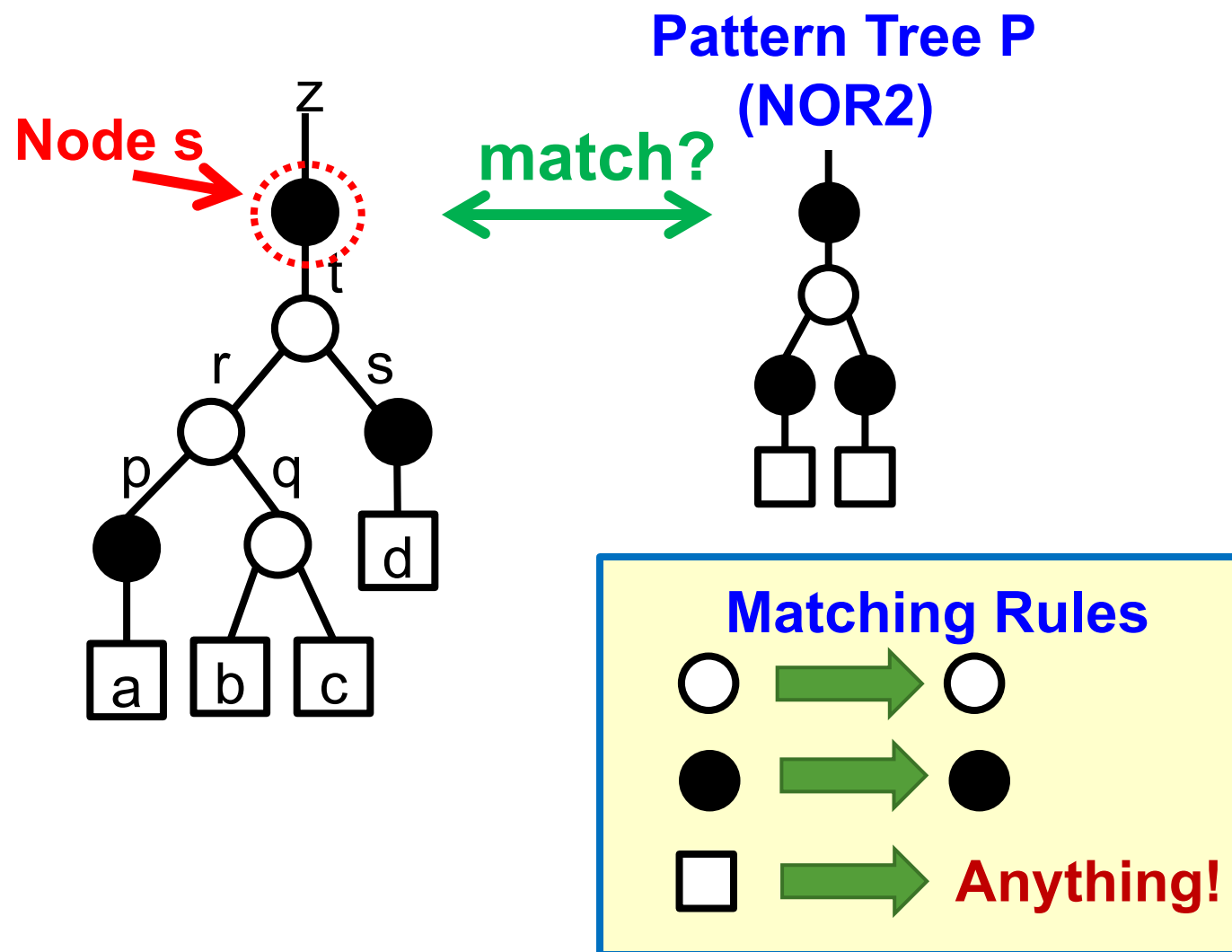
Tree Matching: One Subtlety

- ▶ Be careful matching asymmetric library patterns.
 - One example was AOI21. Need to check all possible matches by “**rotating**” the pattern tree.

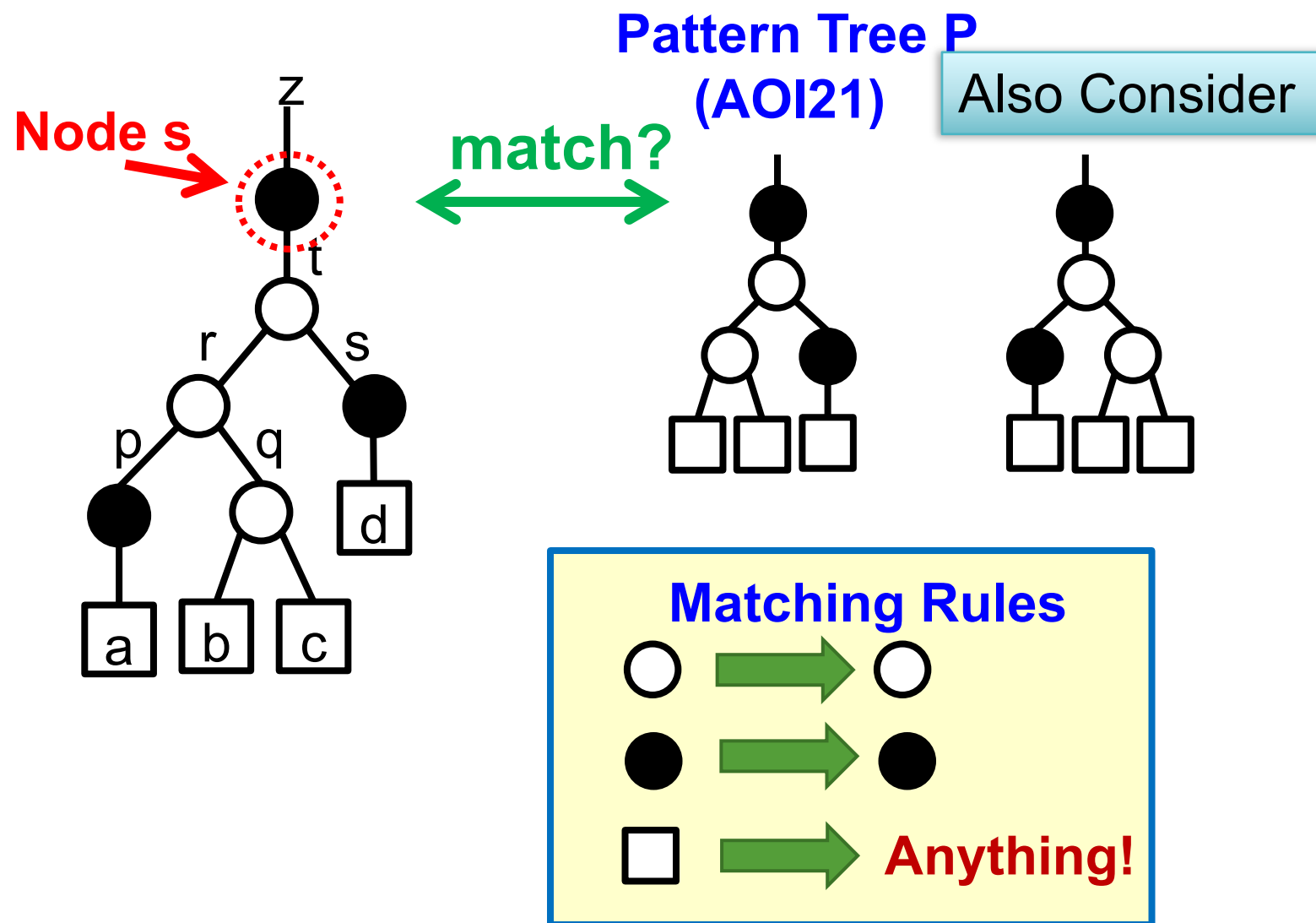


Both orientations are possible!

Tree Matching Example



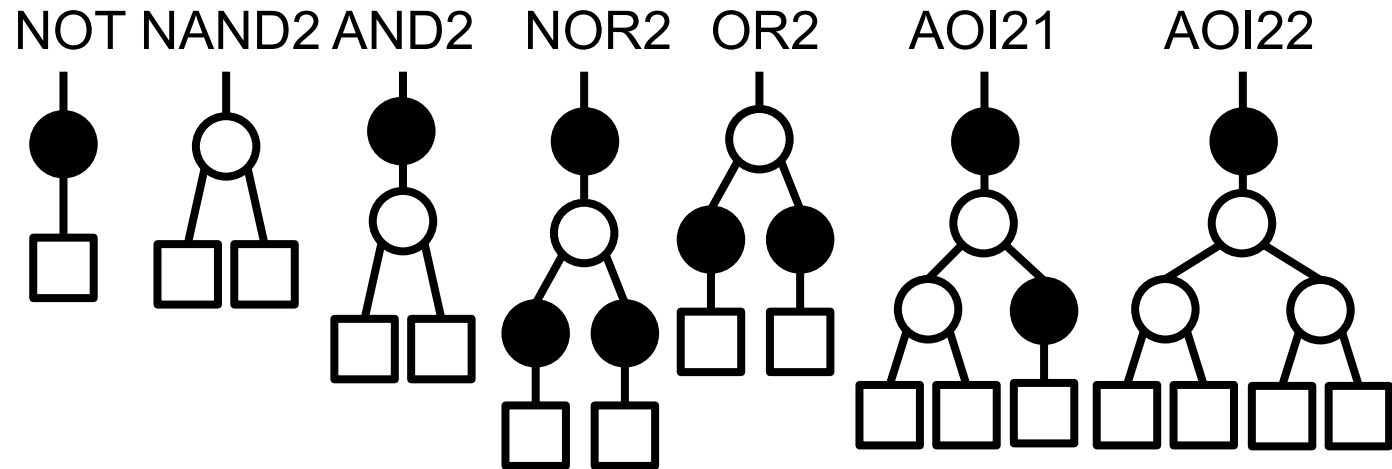
Tree Matching Example



Result After Matching

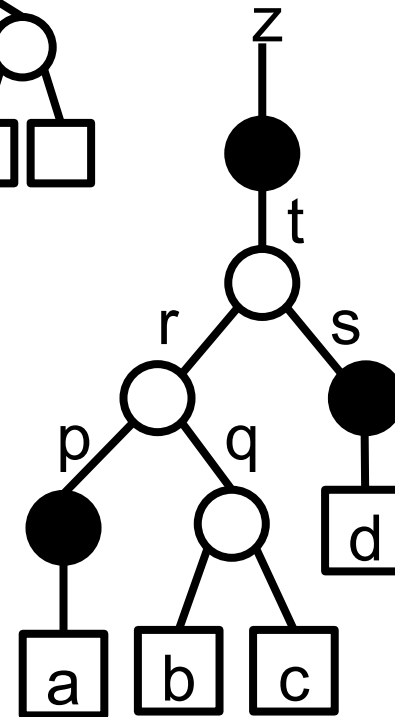
- For each internal node of subject tree, we will get which library pattern trees **match** that node.
- We **annotate** each internal node in the tree with this matching information.

Tree Matching Result Example



➤ List of matching gates

- Node z: {NOT, AND2, AOI21}
- Node t: {NAND2}
- Node r: {NAND2}
- Node s: {NOT}
- Node p: {NOT}
- Node q: {NAND2}



Tech Mapping via Tree Covering

► Subroutines:

- **Treeifying** the input netlist
 - Loses some optimality, but make things simple.
- **Tree matching**
 - For each node in the subject tree, find pattern trees in library that **match**.
- **Minimum-cost covering**
 - Assume you know what can match at each node of subject tree. Then, which ones do you pick for a **minimum cost** cover?

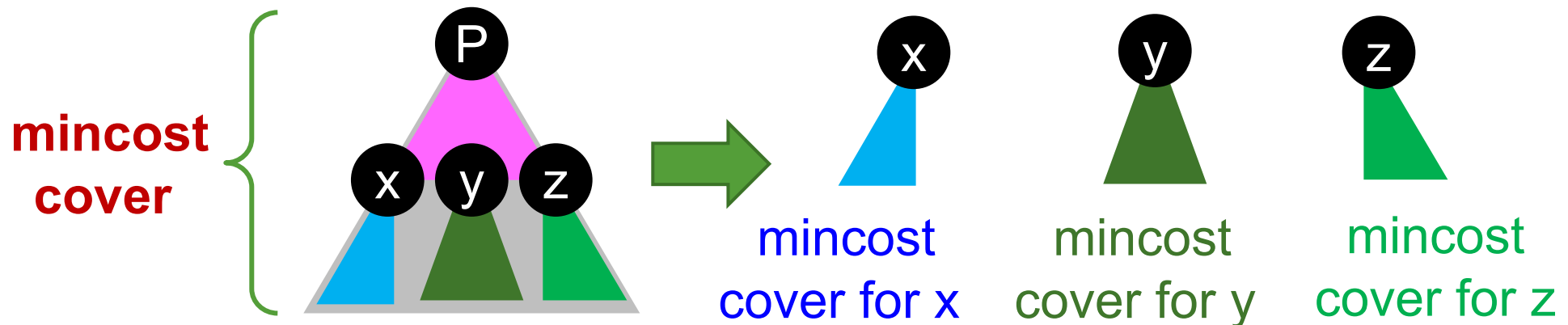
Minimum Cost Covering of Subject Tree

- What cover do we choose?
 - We assign a **cost** to each library pattern.
 - We choose a **minimum cost** (“**mincost**”) cover of the subject tree.

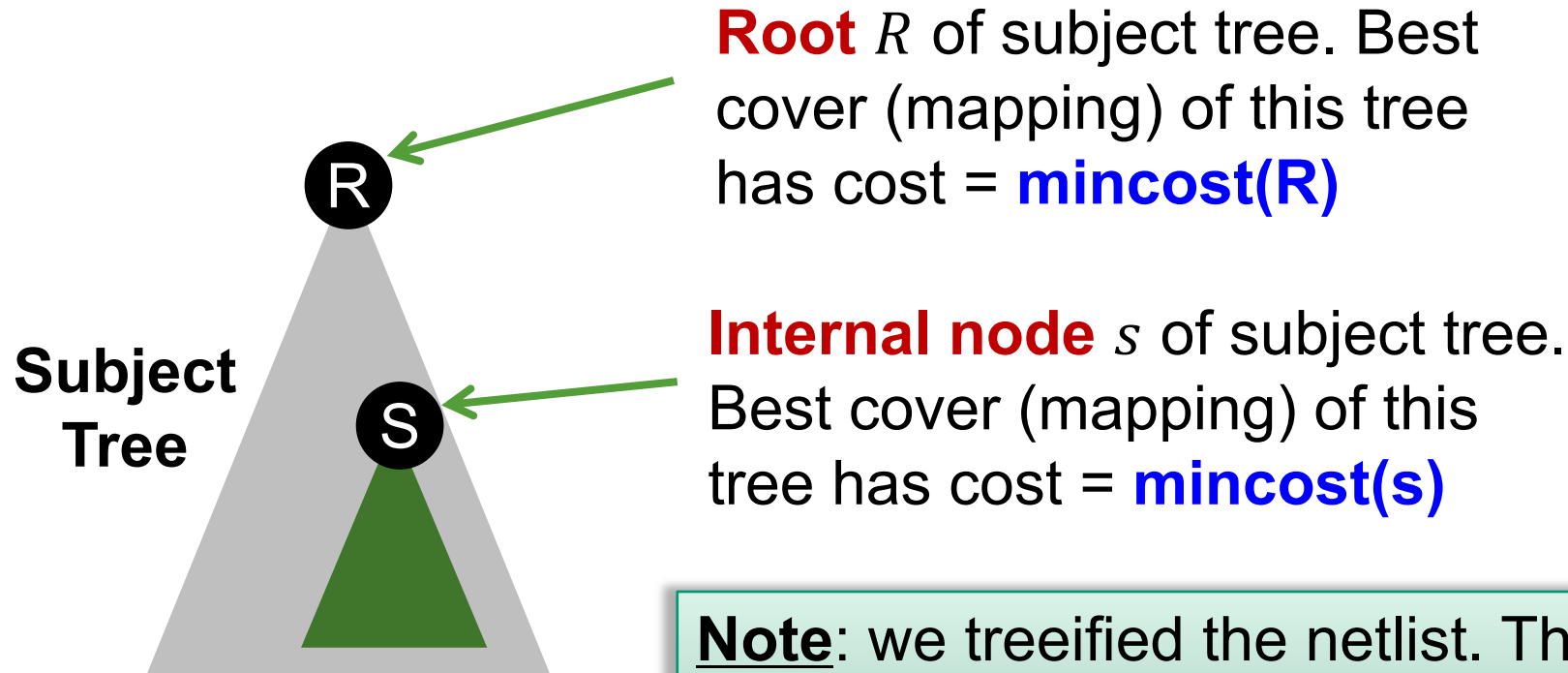
Minimum Cost Covering of Subject Tree

➤ One big idea makes this **easy** to do:

- If pattern P is a **mincost** match at some node s of subject tree, then, **each leaf** of pattern tree must also be the root of some **mincost** matching pattern.
 - Why? By contraposition...
- Leads to a nice recursive algorithm for **mincost** on **any node** in subject tree.
 - This is actually a **dynamic programming** algorithm.



Some Terminology



Note: we treeified the netlist. Thus, every **internal node** (like node s) is the **root** of another, **smaller tree**. This is crucial for our mapping algorithm.

Some Terminology (cont.)

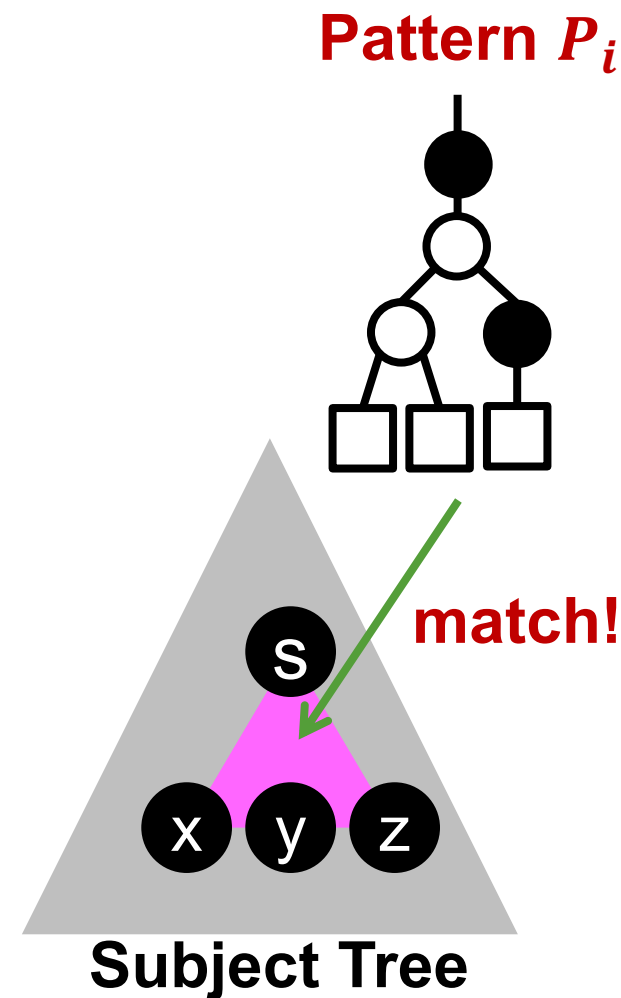
► Suppose:

- Library pattern P_i matches at internal node s of the subject tree.
- Library pattern P_i has m **input nodes**.

► Each of these m “**input nodes**” in library pattern tree P_i will be matched to some nodes in subject tree.

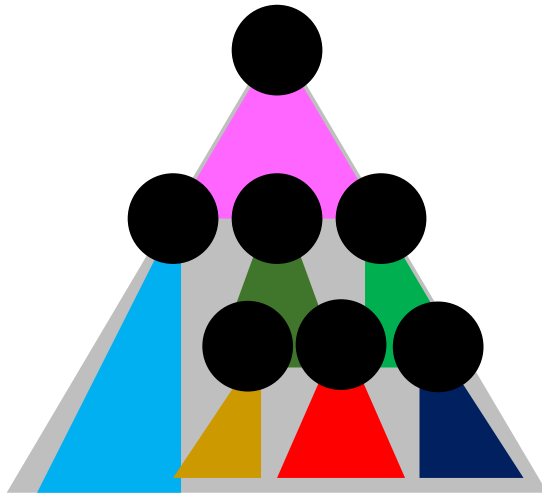
► We call these nodes in the subject tree **leaf nodes** for this matching library pattern tree.

- E.g., x, y, z are leaf nodes.



Calculating Cost of Mapping

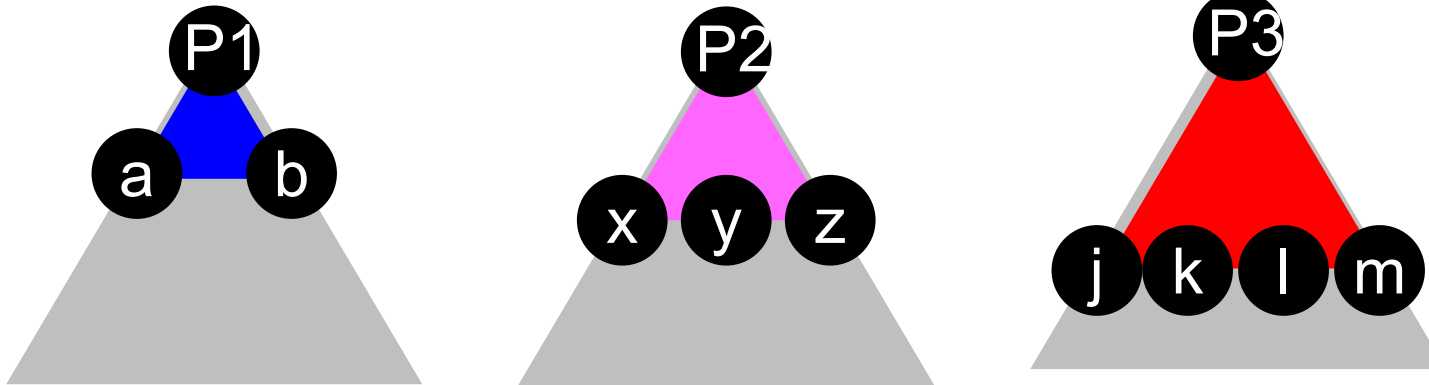
- Every gate pattern in target library has a **cost**.
 - Gate pattern P_i has $\text{cost}(P_i)$.
- To calculate cost of mapping the **entire** subject tree:
 - We add up **cost** for each node where a pattern matches, for all patterns covering subject.



The cost is the sum of the costs of 7 pattern trees.

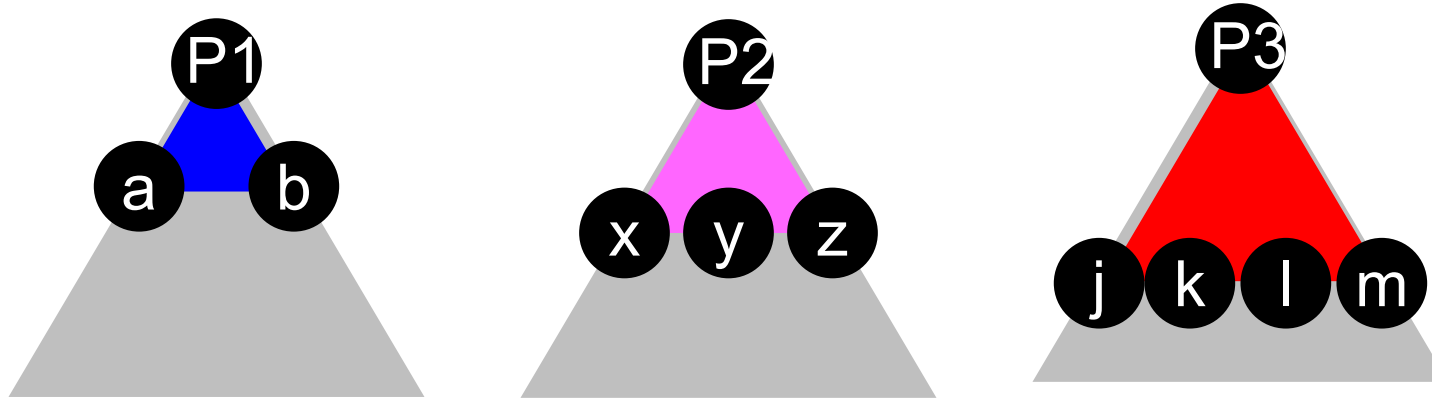
Mincost Tree Covering: The Idea

- Assume 3 **different** library patterns match at root R of subject tree:
 - Pattern $P1$ has 2 leaf nodes: a, b
 - Pattern $P2$ has 3 leaf nodes: x, y, z
 - Pattern $P3$ has 4 leaf nodes: j, k, l, m .



Which of these gates produces the **smallest** value of $\text{mincost}(R)$?

Mincost Tree Cover: The Idea



- Minimum cost of mapping the entire subject tree is

mincost(R)

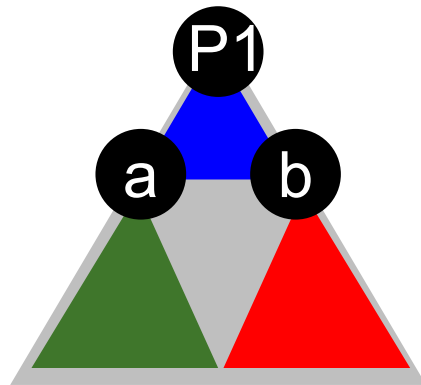
=min{ **minimal** cost with **P1** matching root R,
 minimal cost with **P2** matching root R,
 minimal cost with **P3** matching root R **}**

What are
these?

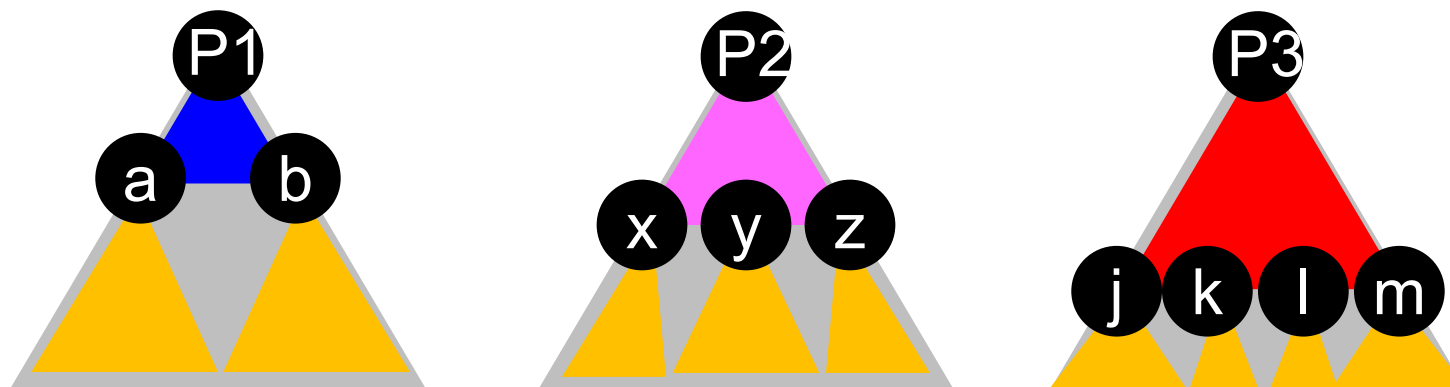
Mincost Tree Cover: Recursive Formula

- In calculating **mincost**(R), we need to answer:
 - What is **minimal** cost with **P1** matching root R?
- Answer: = **cost**(P1) + **mincost**(a) + **mincost**(b)
 - Otherwise, not minimal!

Recursion!



Mincost Tree Cover: Recursive Formula



➔ **mincost(R)**
 = **min**{ **minimal** cost with **P1** matching root R,
 minimal cost with **P2** matching root R,
 minimal cost with **P3** matching root R }

= **min**{ **cost(P1) + mincost(a) + mincost(b),**
 cost(P2) + mincost(x) + mincost(y) + mincost(z),
 cost(P3) + mincost(j) + mincost(k)
 + mincost(l) + mincost(m) }

Mincost Cover: Algorithm

```

mincost( treenode ) {
    cost =  $\infty$ 
    foreach( pattern P matching at subject treenode ) {
        let L = {nodes in subject tree corresponding to leaf nodes in P
                when P is placed with its root at treenode }
        newcost = cost(P)
        foreach( node n in L ) {
            newcost = newcost + mincost( n );
        }
        if ( newcost < cost ) then {
            cost = newcost;
            treenode.BestLibPattern = P;
        }
    }
}

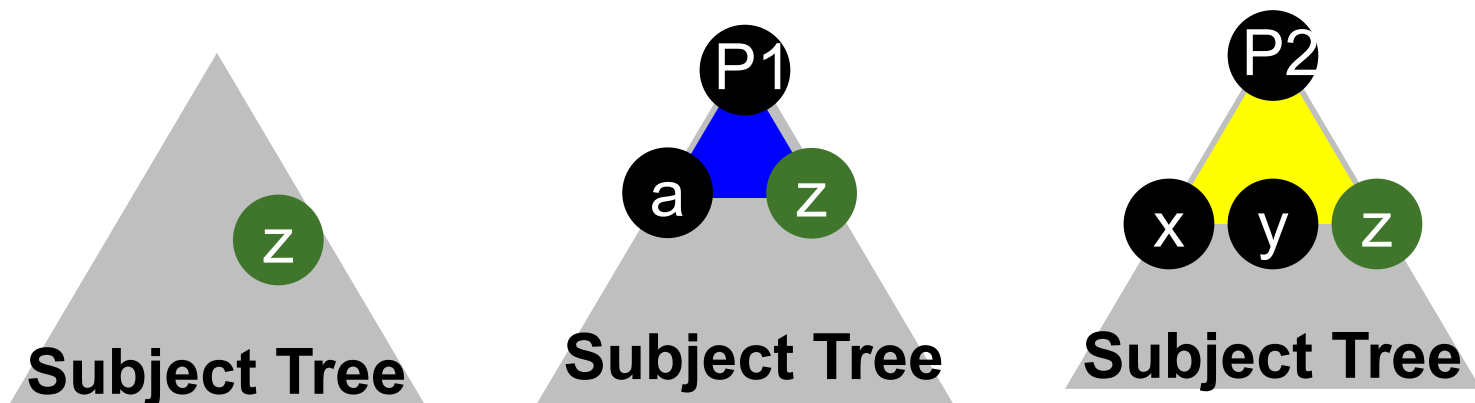
```

Min Cost Tree Cover

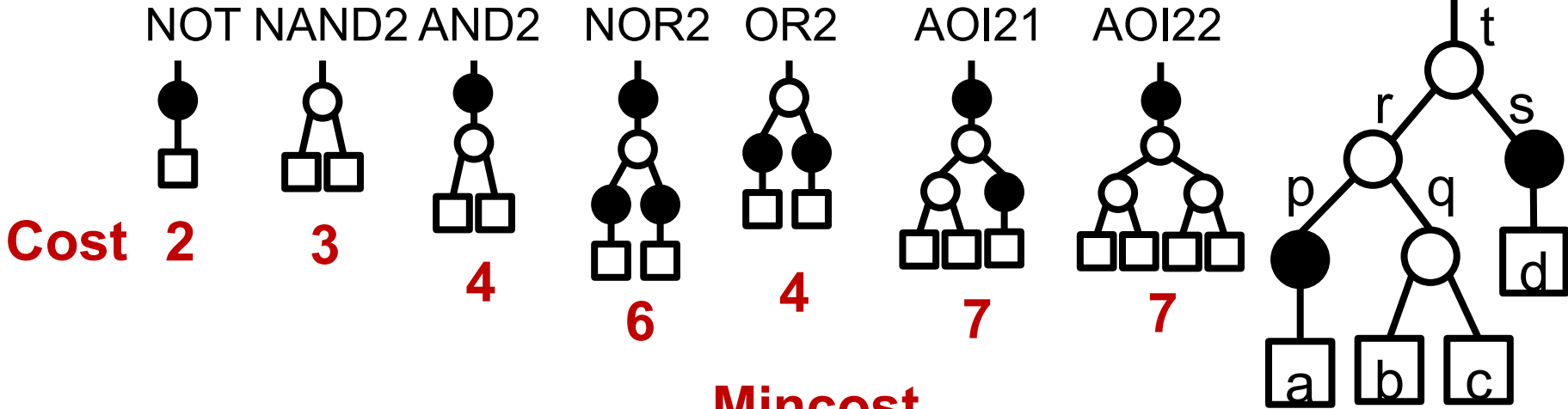
- One **redundant** computation we must note:
 - This algorithm will **revisit** same tree node many times during recursions...
 - ...and it will **recompute** the mincost cover for that node each time.
- Can we do better...?
 - Yes, just keep a table with **mincost** value for each node.
 - Start with value ∞ and when node's cost gets computed, this value gets **updated**.
 - Each time computing **mincost(node)**, **check first** to see if **node** has been visited before computing it-- saves computation!

Illustration

- Node “z” in this subject tree
 - will get its **mincost(z)** cover computed when we put **P1** at root of subject tree...
 - ...and again when we put **P2** at the root.
 - **Better solution:** just compute it **once**, first time, **save** it, and **look it up** later!



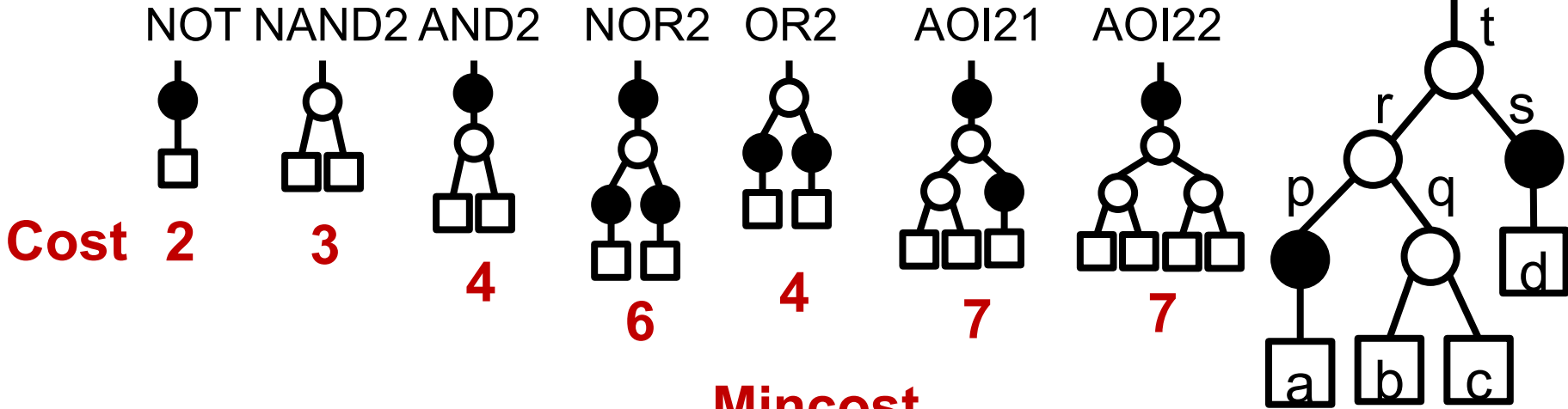
Min Cost Tree Cover Example



Node	Can Match	Mincost
z	$\left\{ \begin{array}{l} \text{NOT} \\ \text{AND2} \\ \text{AOI21} \end{array} \right\}$	$\left\{ \begin{array}{l} 2 + \text{mincost}(t) \\ 4 + \text{mincost}(r) + \text{mincost}(s) \\ 7 + \text{mincost}(p) + \text{mincost}(q) \end{array} \right\}$
t	NAND2	$3 + \text{mincost}(r) + \text{mincost}(s)$
r	NAND2	$3 + \text{mincost}(p) + \text{mincost}(q)$
p	NOT	$2 \Rightarrow \text{mincost}(p) = 2$
q	NAND2	$3 \Rightarrow \text{mincost}(q) = 3$
s	NOT	$2 \Rightarrow \text{mincost}(s) = 2$

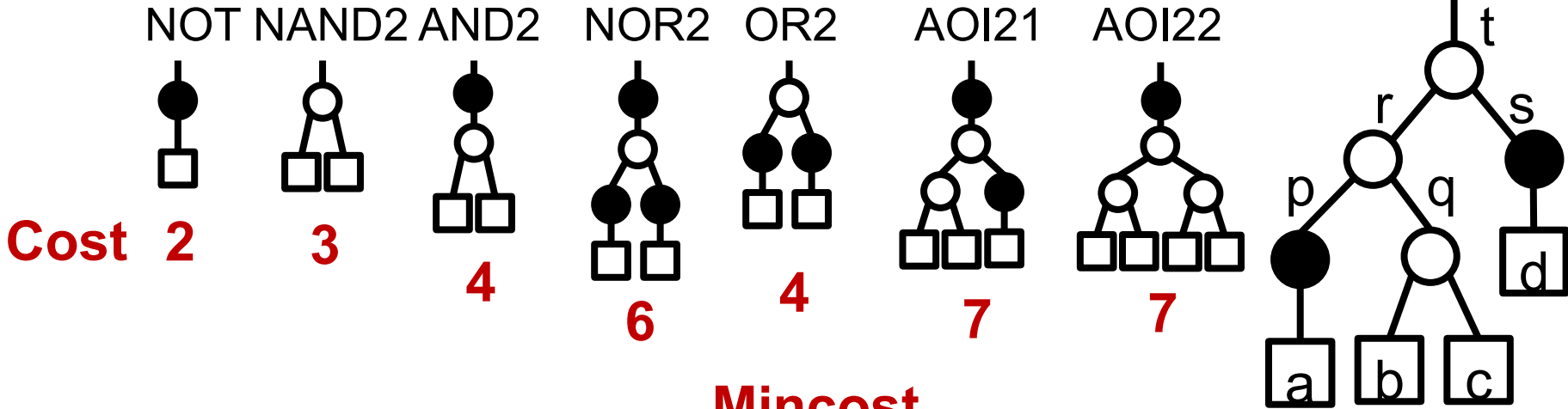
min

Min Cost Tree Cover Example



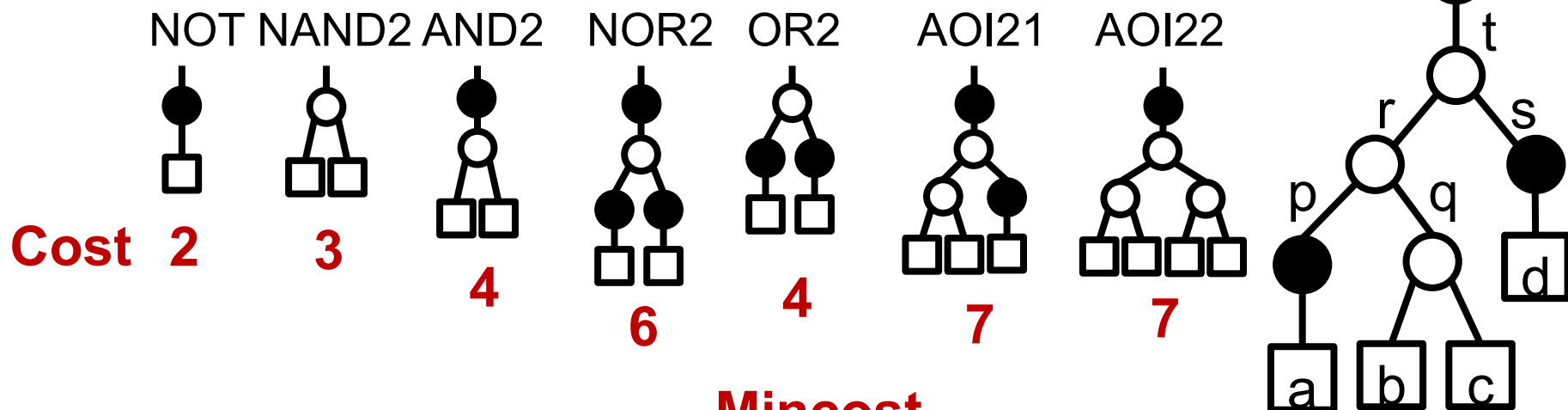
Node	Can Match	Mincost	
z	$\left\{ \begin{array}{l} \text{NOT} \\ \text{AND2} \\ \text{AOI21} \end{array} \right\}$	$\left\{ \begin{array}{l} 2 + \text{mincost}(t) \\ 4 + \text{mincost}(r) + \text{mincost}(s) \\ 7 + \text{mincost}(p) + \text{mincost}(q) \end{array} \right\}$	$\Rightarrow \text{mincost} = 12$
t	NAND2	$3 + \text{mincost}(r) + \text{mincost}(s)$	
r	NAND2	$3 + \text{mincost}(p) + \text{mincost}(q)$	$\Rightarrow \text{mincost}(r) = 8$
p	NOT	$2 \Rightarrow \text{mincost}(p) = 2$	
q	NAND2	$3 \Rightarrow \text{mincost}(q) = 3$	
s	NOT	$2 \Rightarrow \text{mincost}(s) = 2$	

Min Cost Tree Cover Example



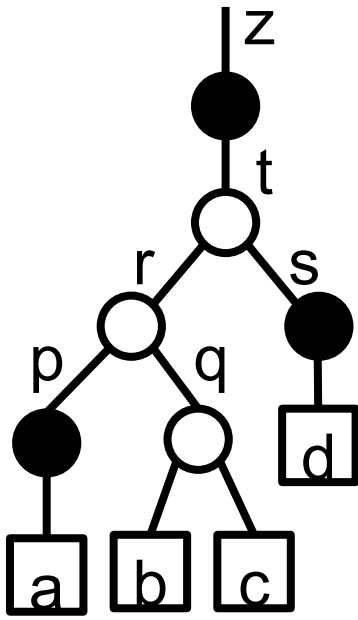
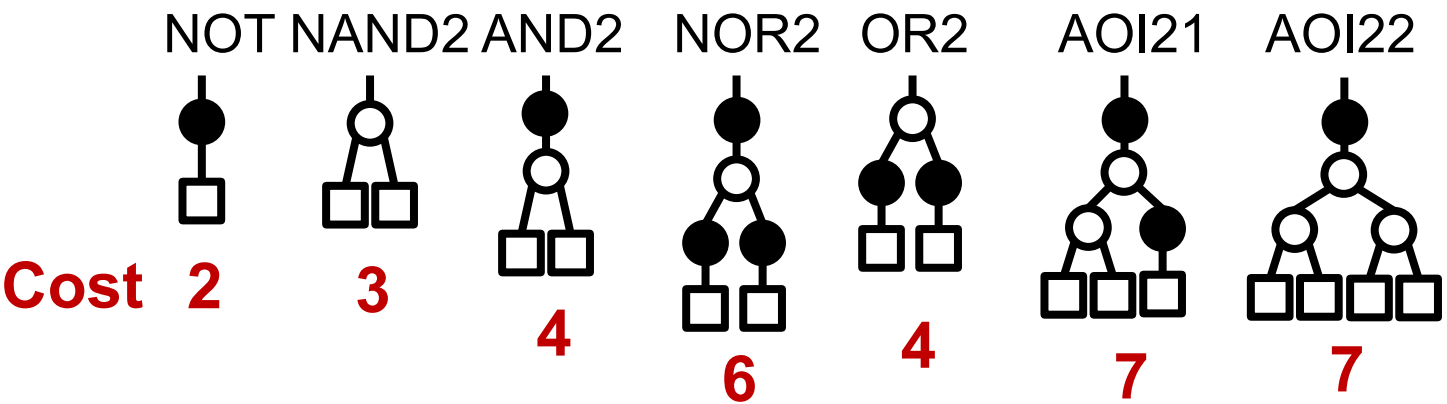
Node	Can Match	Mincost	
z	{ NOT AND2 AOI21 }	$2 + \text{mincost}(t)$ $4 + \text{mincost}(r) + \text{mincost}(s)$ $7 + \text{mincost}(p) + \text{mincost}(q)$	$\Rightarrow \text{mincost} = 14$ $\Rightarrow \text{mincost} = 12$
t	NAND2	$3 + \text{mincost}(r) + \text{mincost}(s)$	$\Rightarrow \text{mincost}(t) = 13$
r	NAND2	$3 + \text{mincost}(p) + \text{mincost}(q)$	$\Rightarrow \text{mincost}(r) = 8$
p	NOT	$2 \Rightarrow \text{mincost}(p) = 2$	
q	NAND2	$3 \Rightarrow \text{mincost}(q) = 3$	
s	NOT	$2 \Rightarrow \text{mincost}(s) = 2$	

Min Cost Tree Cover Example



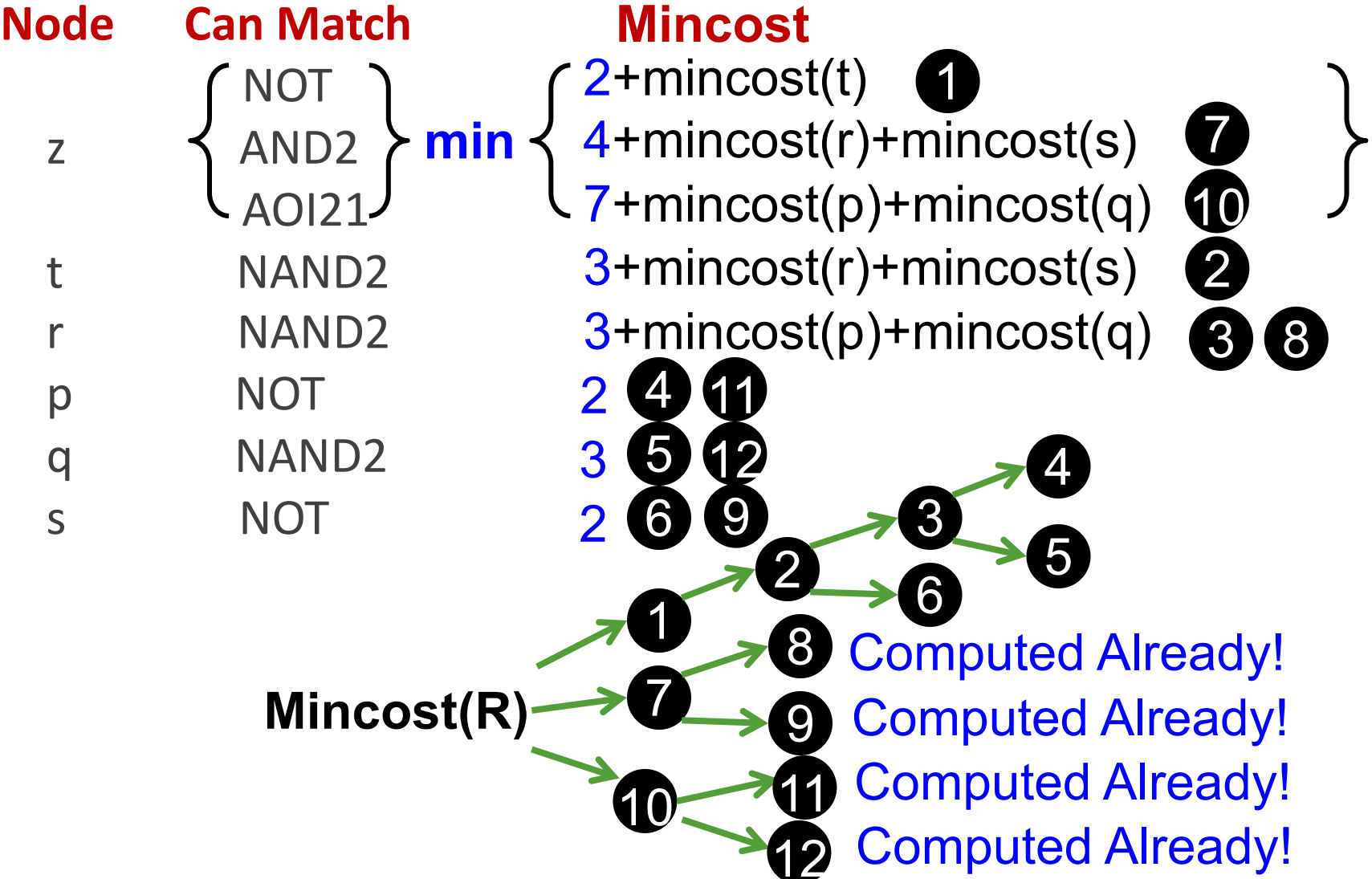
Node	Can Match	Mincost	
z	$\left\{ \begin{array}{l} \text{NOT} \\ \text{AND2} \\ \text{AOI21} \end{array} \right\}$	$\left\{ \begin{array}{l} 2 + \text{mincost}(t) \\ 4 + \text{mincost}(r) + \text{mincost}(s) \\ 7 + \text{mincost}(p) + \text{mincost}(q) \end{array} \right\}$	$\Rightarrow \text{mincost} = 15$ min $\Rightarrow \text{mincost} = 14$ $\Rightarrow \text{mincost} = 12$
t	NAND2	$3 + \text{mincost}(r) + \text{mincost}(s)$	$\Rightarrow \text{mincost}(t) = 13$
r	NAND2	$3 + \text{mincost}(p) + \text{mincost}(q)$	$\Rightarrow \text{mincost}(r) = 8$
p	NOT	$2 \Rightarrow \text{mincost}(p) = 2$	
q	NAND2	$3 \Rightarrow \text{mincost}(q) = 3$	
s	NOT	$2 \Rightarrow \text{mincost}(s) = 2$	

Min Cost Tree Cover Example



Node	Can Match	Mincost	
z	{ NOT AND2 }	15	min
		14	
	{ AOI21 }	12	
t	NAND2	13	Best!
r	NAND2	8	
p	NOT	2	
q	NAND2	3	
s	NOT	2	

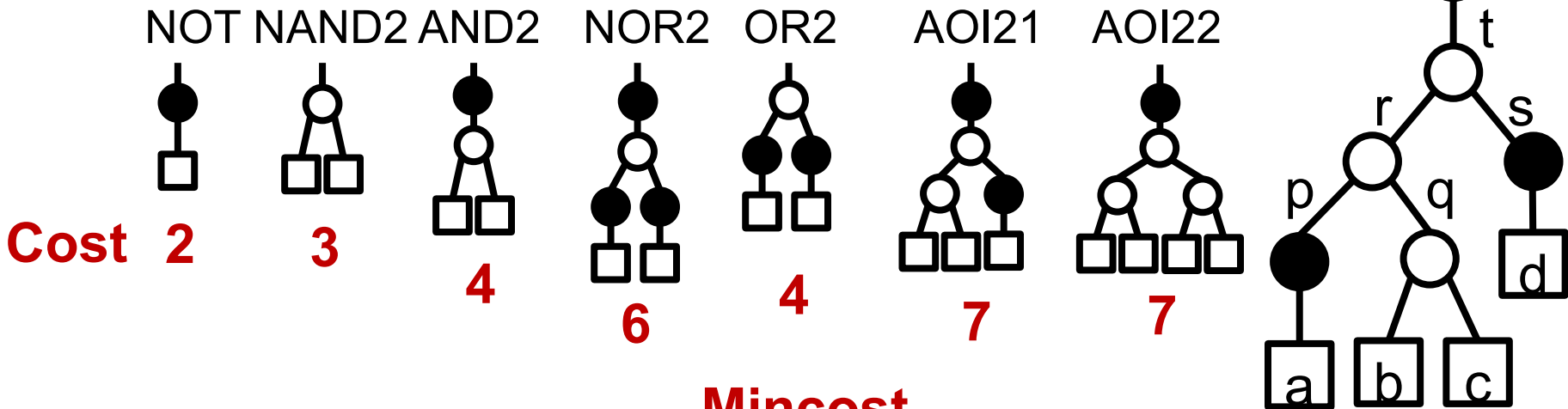
Actual Execution Sequence



Min Cost Cover: How To Get Final Cover?

- Look at **best cost** at subject root. Find pattern P at the root that gives the cost.
- Find **leaf nodes** in the subject tree for pattern P .
 - Look at the **best cost** at each of these leaf nodes.
 - Find the pattern P_i that is associated with each of these best costs.
 - Look again at leaf nodes in the subject tree that are associated with each of these patterns P_i .
 - Repeat...

Min Cost Tree Cover Example



Node	Can Match	Mincost
z	<div> NOT AND2 AOI21 </div>	<div> $2 + \text{mincost}(t) = 15$ $4 + \text{mincost}(r) + \text{mincost}(s) = 14$ $7 + \text{mincost}(p) + \text{mincost}(q) = 12$ </div>
t	NAND2	$3 + \text{mincost}(r) + \text{mincost}(s) = 13$
r	NAND2	$3 + \text{mincost}(p) + \text{mincost}(q) = 8$
p	NOT	2
q	NAND2	3
s	NOT	2

min
Best!

Min Cost Tree Cover

- Turns out to be several nice **extensions** possible
 - Can modify algorithm a little to minimize **delay** instead of cost.
 - Many interesting and useful variations, starting from this algorithm skeleton.

Technology Mapping: Summary

- Synthesis gives you “**uncommitted**” or “technology independent” design, e.g., NAND2 and NOT.
- Technology mapping turns this into **real gates** from library.
 - Can determine difference between good and bad implementations.
- Tree covering
 - One nice, simple, elegant approach to the problem.
 - 3 parts: treeify input, match all library patterns, find min cost cover.
- There are other ways to do this. Some work with real Boolean algebra in mapping.
- Has other applications, like for Lookup-Table (LUT) FPGA.
 - With different algorithm.

Outline

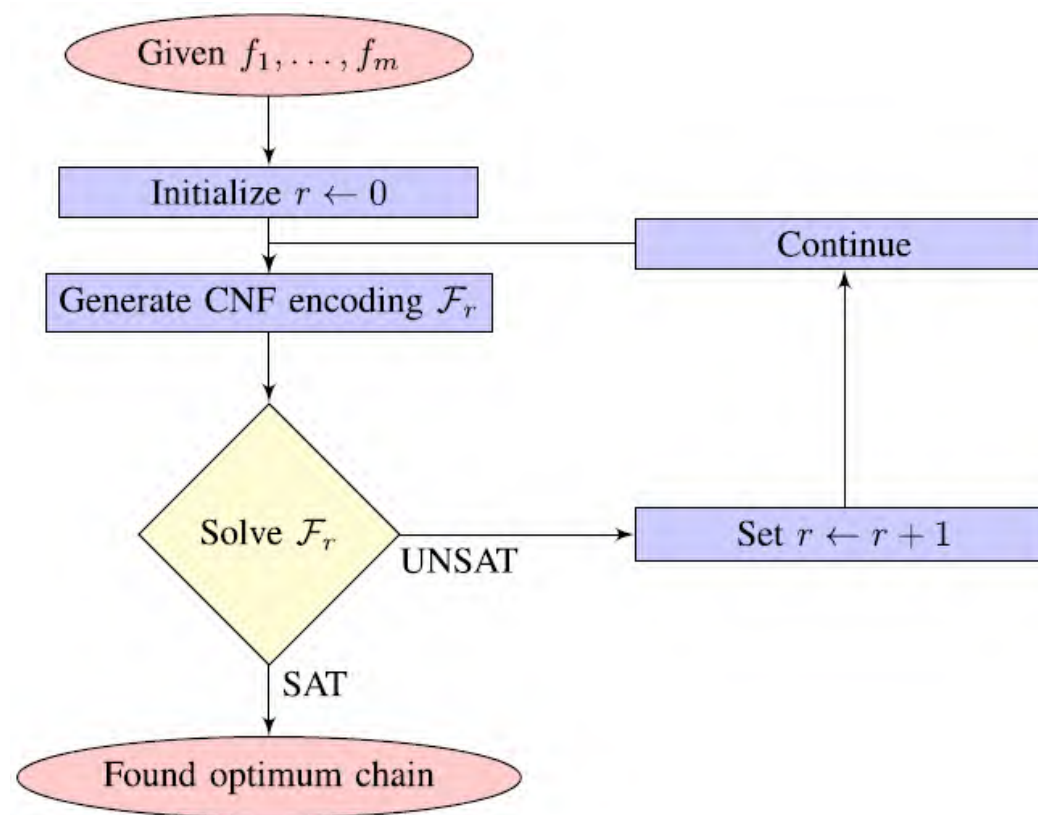
- Background on Logic Synthesis
- Technology-Independent Synthesis
- Technology Mapping
- Exact Synthesis

Exact Synthesis

Exact synthesis

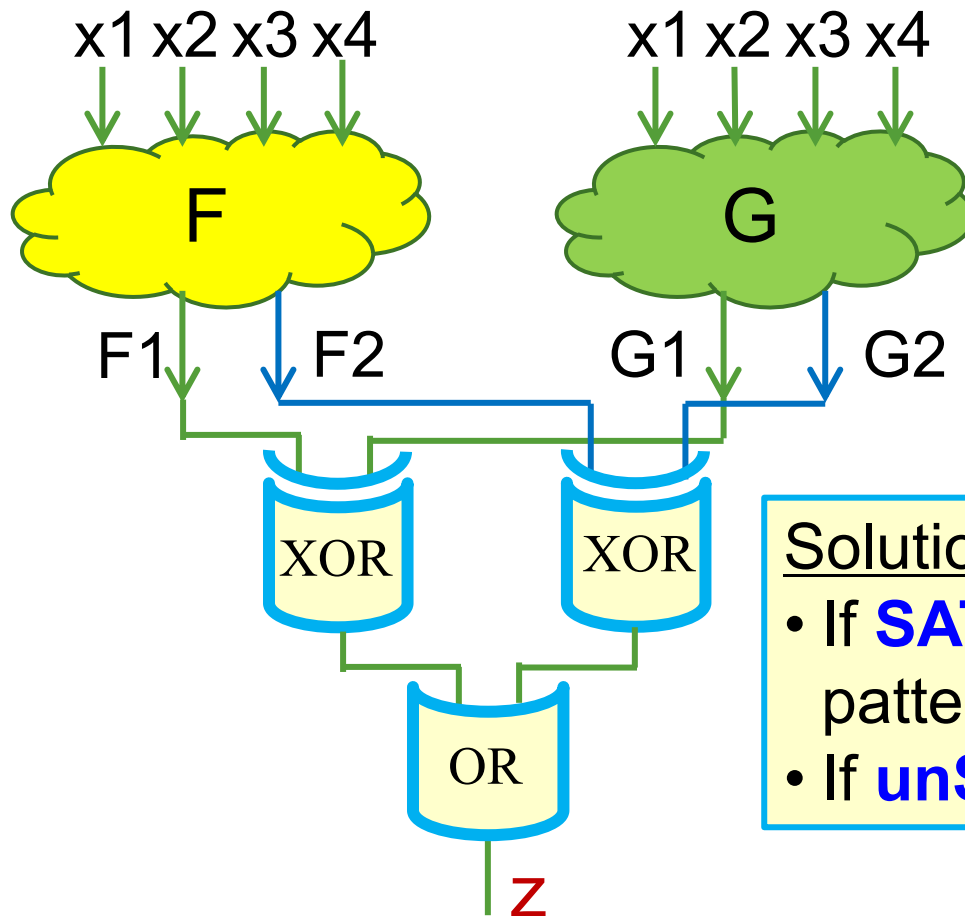
- Given a function f , does there exist a logic network N with exactly r gates implementing f ?

- Can be applied to synthesize a circuit with fewest gates.



Application of SAT in EDA

- Do these two logic networks implement the **same** Boolean function?

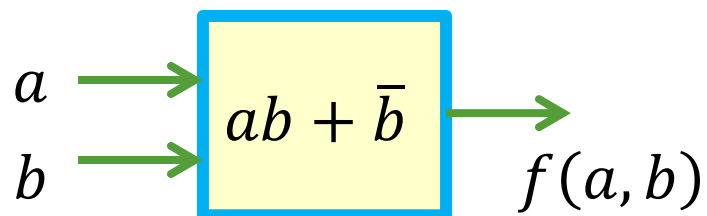


Solution: Do **SAT** on this new network

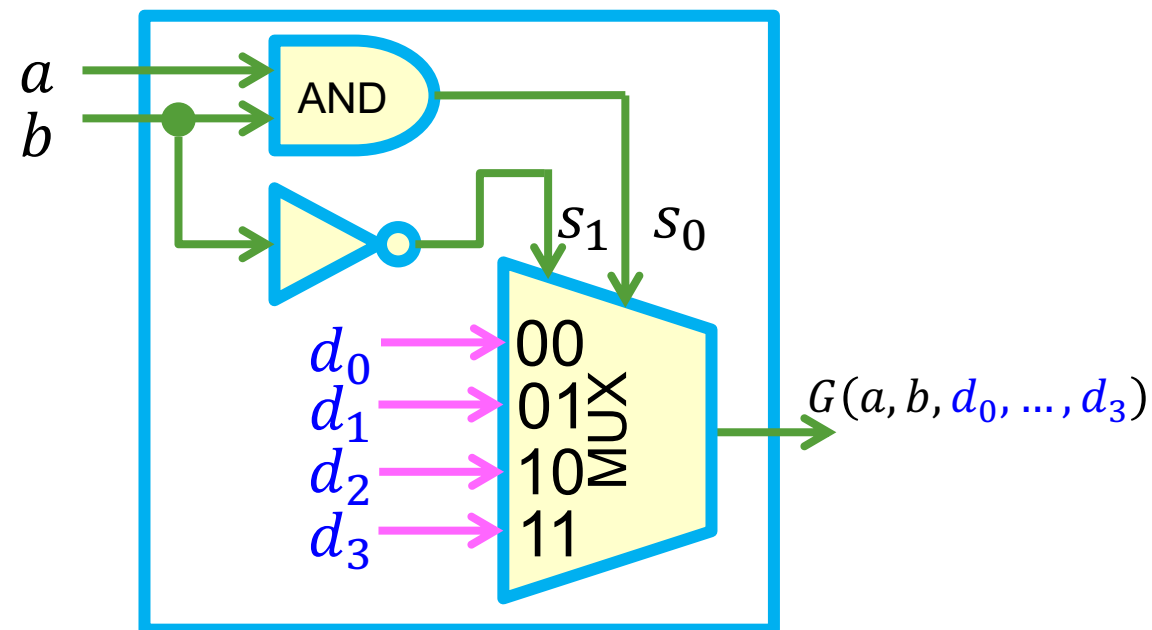
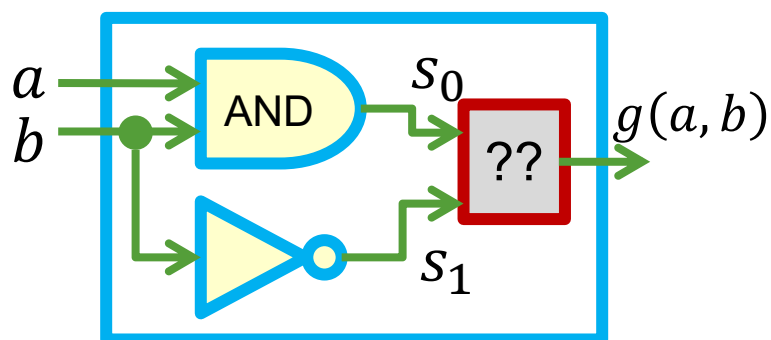
- If **SAT**: networks not same, and this pattern makes them give different outputs.
- If **unSAT**: yes, same!

Example: Network Repair

Specified



Implemented



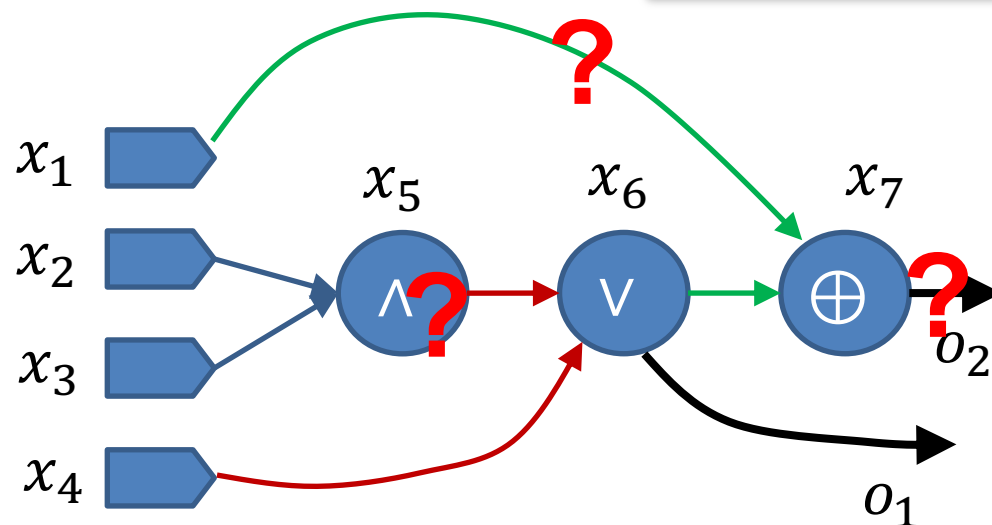
- We want to find (d_0, d_1, d_2, d_3) so that $(\forall ab z)(d_0, d_1, d_2, d_3) = 1$
- To repair the network, we only need **one satisfying assignment** for (d_0, d_1, d_2, d_3) .
- If **unSAT**, the network repair is impossible!

Model: Boolean Chain

- Assume n primary inputs (PIs), m outputs, and r gates
 - PIs: x_1, \dots, x_n
 - Gates: $x_{n+1}, x_{n+2}, \dots, x_{n+r}$
 - Assume each gate is a K -input gate

Boolean Chain Example

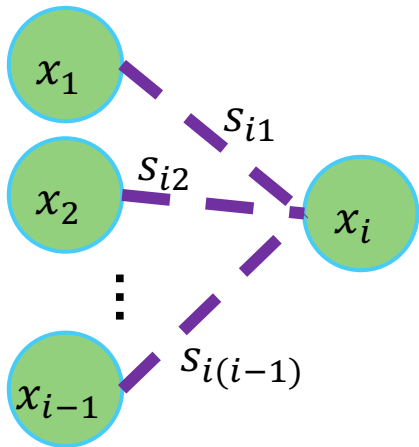
$K = 2$ for our illustration



To be determined: connections, gate functions, and outputs

Variables

s_{ij} : whether node x_i has input from node x_j



x_{it} : global function of node x_i under combination t

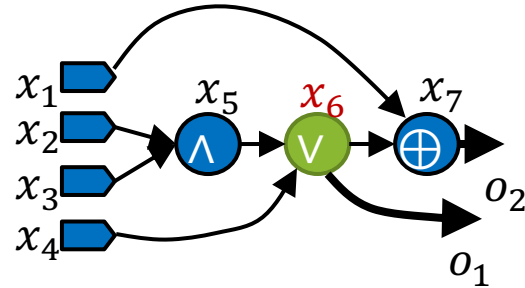
t	x_{jt}	x_{kt}	x_{it}
00...00	x_{j0}	x_{k0}	x_{i0}
00...01	x_{j1}	x_{k1}	x_{i1}
\vdots			
11...11	x_{jN}	x_{kN}	x_{iN}

($N = 2^n - 1$)

f_{ipq} : local function of node x_i

(p, q)	f_{ipq}
00	f_{i00}
01	f_{i01}
10	f_{i10}
11	f_{i11}

- $s_{61} = 0$
- $s_{62} = 0$
- $s_{63} = 0$
- $s_{64} = 1$
- $s_{65} = 1$



global function of x_6

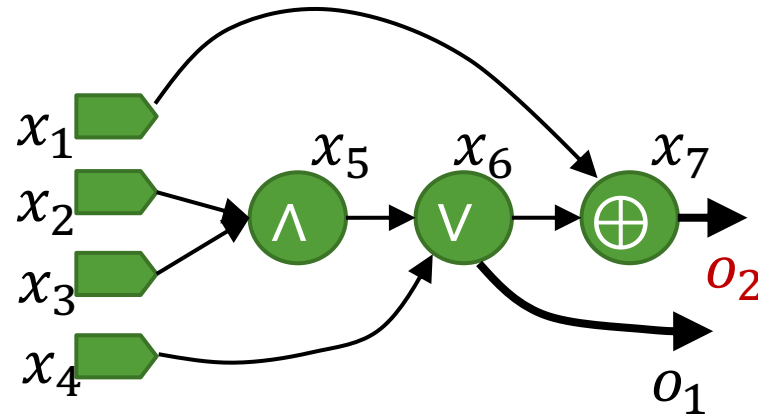
t	x_{6t}
0000	0
0001	1
\vdots	
1111	1

local function of x_6

(p, q)	f_{6pq}
00	0
01	1
10	1
11	1

Variables

g_{hi} : whether the h -th ($1 \leq h \leq m$) output is given by node x_i

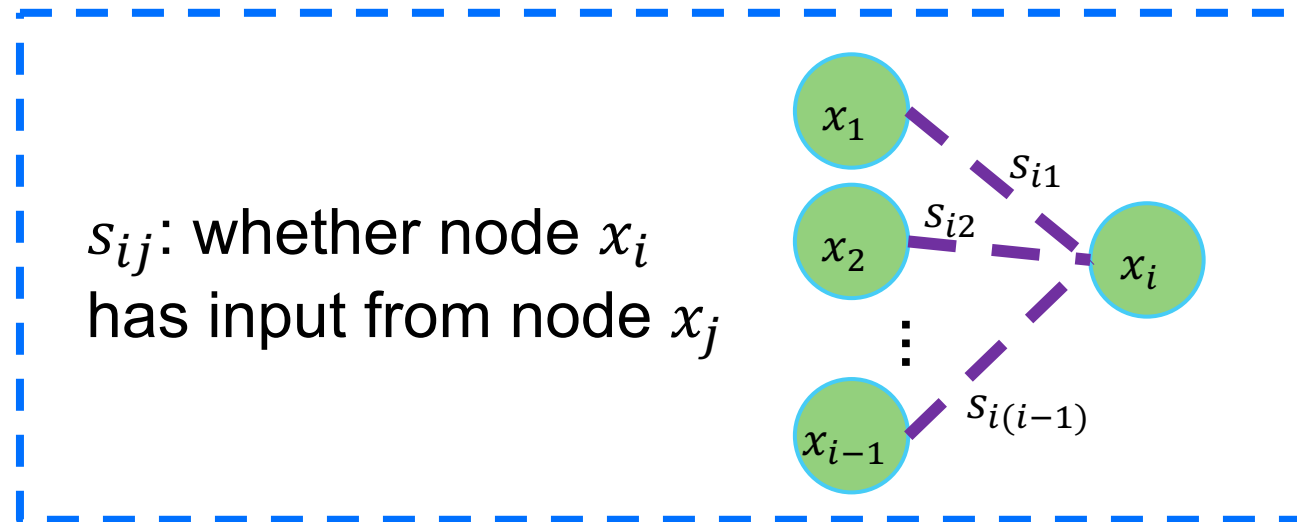


$$\begin{aligned}
 g_{21} &= 0 \\
 &\vdots \\
 g_{26} &= 0 \\
 g_{27} &= 1
 \end{aligned}$$

Fanin Number Constraint

- Each node has exactly K inputs: For $n + 1 \leq i \leq n + r$,

$$\sum_{j=1}^{i-1} s_{ij} = K$$



Output Constraints

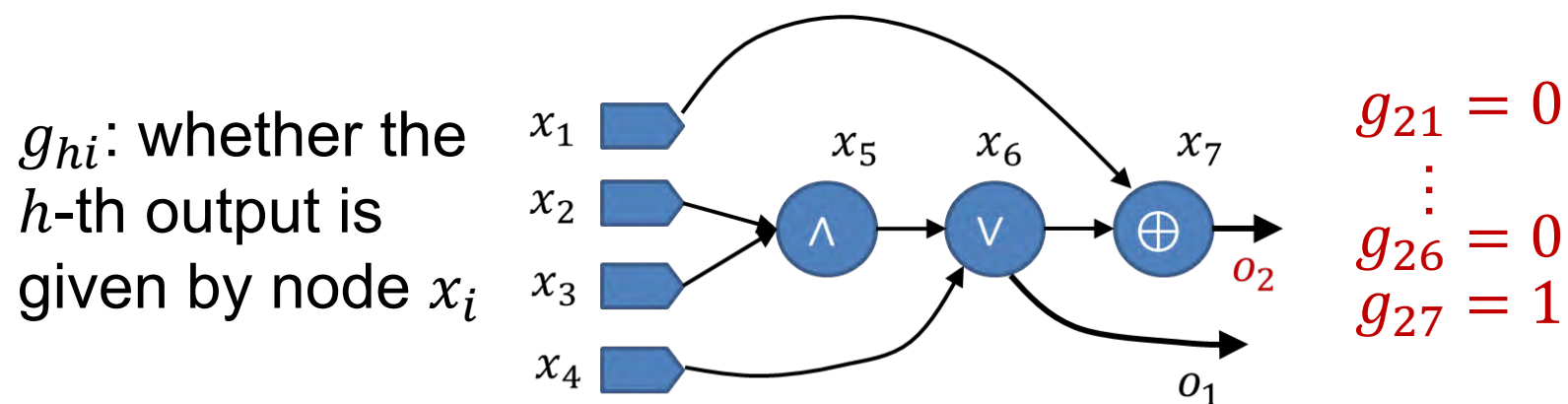
- Suppose the h -th output's global function is o_{ht}

- Constraint 1 [**One g_{hi} is 1**]: For $1 \leq h \leq m$,

$$\sum_{i=1}^{n+r} g_{hi} = 1$$

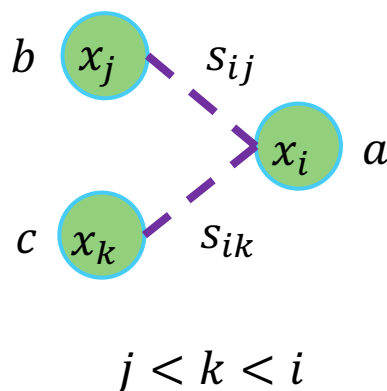
- Constraint 2 [**Global function equals output function**]:

For $1 \leq h \leq m, 1 \leq i \leq n + r, 0 \leq t < 2^n, g_{hi} \Rightarrow (x_{it} = o_{ht})$



Connection Constraint

s_{ij} : whether node x_i has input from node x_j



x_{it} : global function of node x_i under combination t

t	x_{jt}	x_{kt}	x_{it}
00...00	x_{j0}	x_{k0}	x_{i0}
00...01	x_{j1}	x_{k1}	x_{i1}
\vdots			
11...11	x_{jN}	x_{kN}	x_{iN}

$$(N = 2^n - 1)$$

f_{ipq} : local function of node x_i

(p, q)	f_{ipq}
00	f_{i00}
01	f_{i01}
10	f_{i10}
11	f_{i11}

- Circuit encoded by the variables is legal and gives the correct output

$$\left(s_{ij} \wedge s_{ik} \wedge (x_{it} = a) \wedge (x_{jt} = b) \wedge (x_{kt} = c) \right) \Rightarrow (f_{ibc} = a)$$

for all $1 \leq i \leq n + r, 1 \leq j < k < i, 0 < t < 2^n, a, b, c \in \{0, 1\}$

Exact Synthesis: Summary

- Boolean variables
 - Connection, global function, local function, and output binding
- Constraints
 - Fanin number constraint
 - Output constraints
 - Connection constraint
- For small-scale designs, can find exact solution in terms of number of k -input gates
 - Can be adapted to optimize area and delay
- For large-scale designs
 - Use exact synthesis for local sub-circuits
 - Precompute to reduce runtime!

Final Summary

- Two phases in multi-level logic synthesis:
 - Technology-independent synthesis
 - Technology mapping
- Technology-independent synthesis
 - We studied: Boolean logic network and AIG
 - More: majority-inverter graph (MIG), XOR-majority graph (XMG)
 - Mockturtle: <https://github.com/lsils/mockturtle>
- Technology mapping: tree mapping algorithm
- Exact synthesis: formulate SAT problems to find optimal designs