ECE6703J

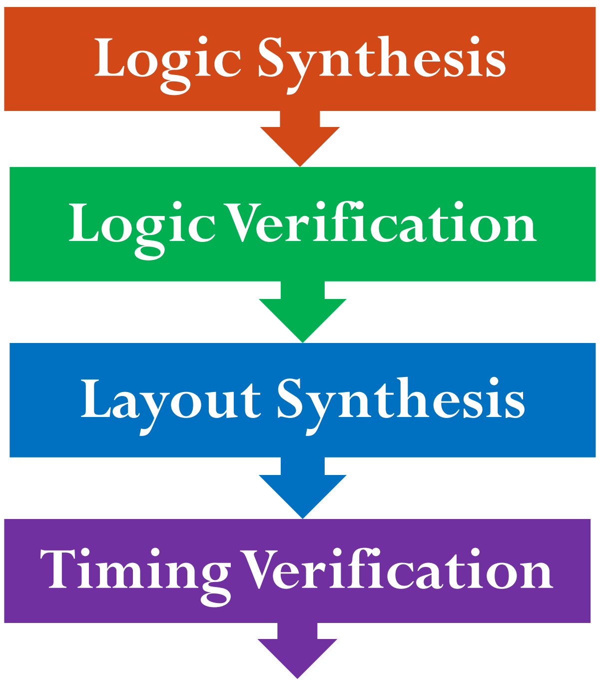
Computer-Aided Design of Integrated Circuits

Technology Mapping

Outline

- Technology mapping
  - Problem
  - Background on the Solution -- Tree Covering Algorithm
  - Details of Tree Covering Algorithm

The Focus of Our Course



- Start with some Boolean / logic design description …
- …end with gates+wires, located at (x,y) coordinates on chip

Logic Synthesis

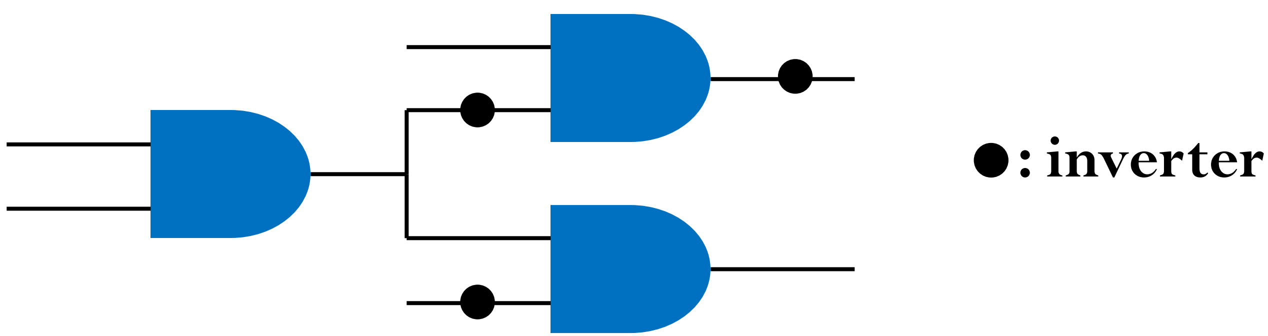
High-level design description → **Logic Synthesis** → Connected gates + wires in your technology. Called "netlist"

- Two steps in logic synthesis:
  - Technology-independent synthesis
  - Technology mapping

Technology-Independent Synthesis

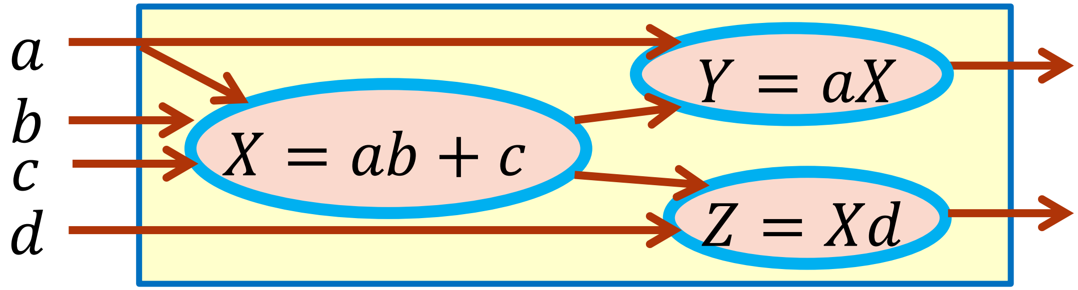
- Output: **An abstract representation of Boolean function**
  - It is **not** the actual **gate-level netlist**.
  - The result is called: "**uncommitted**" logic, or "**technology independent**" logic.
  - Example: **AND-inverter graph (AIG)**, **Boolean logic network**, etc.

AND-inverter graph (AIG)



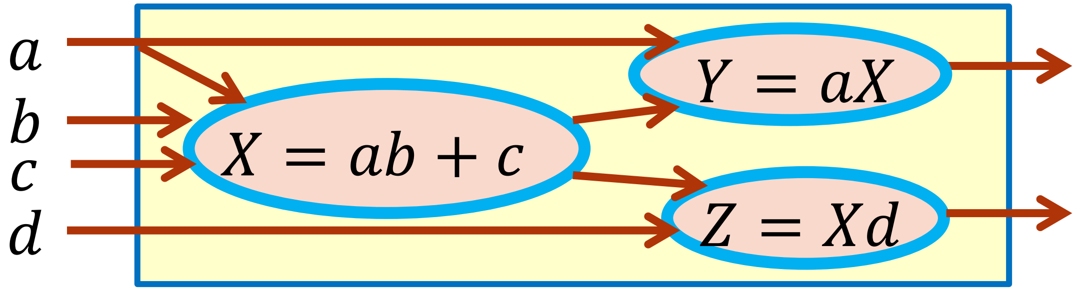
Boolean Logic Network

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



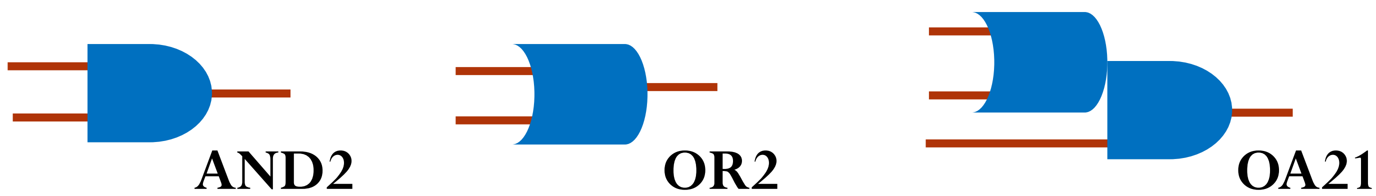
Tech Mapping: The Problem

- Boolean logic network is still a little **abstract**.
  - We know the network structure and the Boolean function of each node.
  - … but that still does **not** give us the actual **gate-level netlist**.

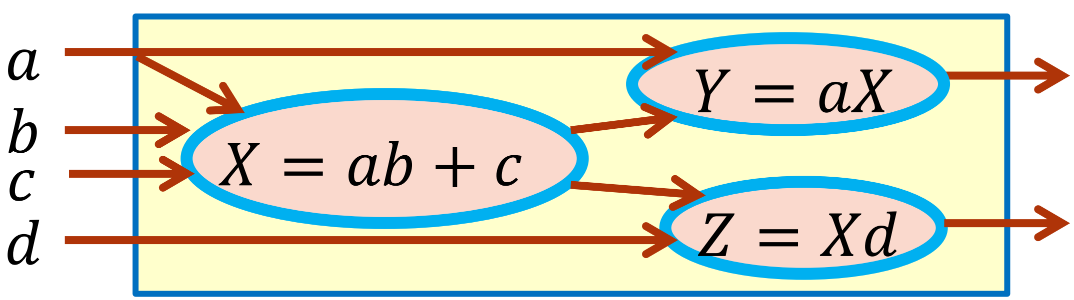


Tech Mapping: The Problem

- Usually, we are given a gate library, from which we can pick gates. Suppose we have these gates in our "**library**".



  - This is called "**the technology**" we are allowed to use to build the 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

OAI22

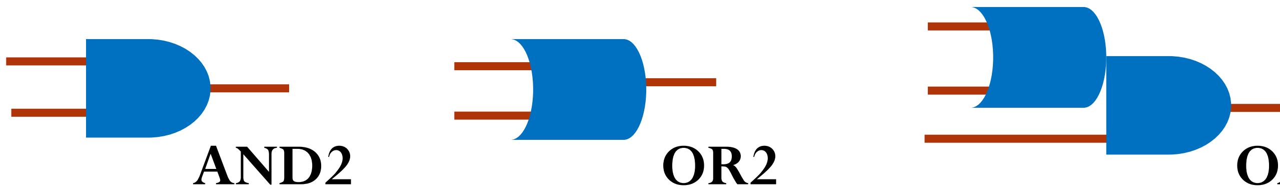
AND-OR-Inverter

AOI221

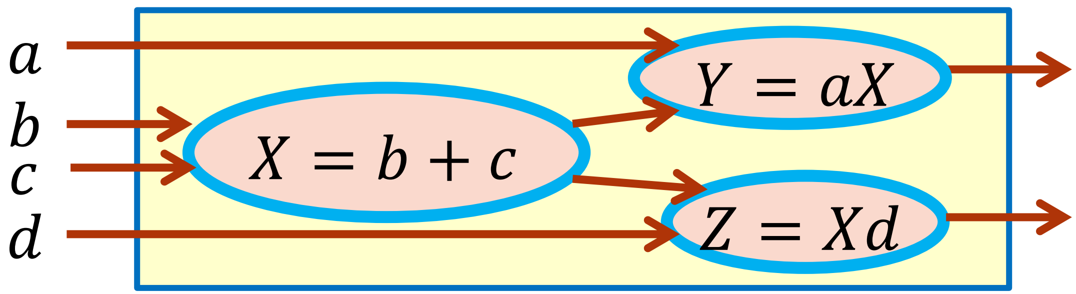
- In CMOS, OAI and AOI gate structures are **efficient** at transistor level.

Tech Mapping: Simple Example

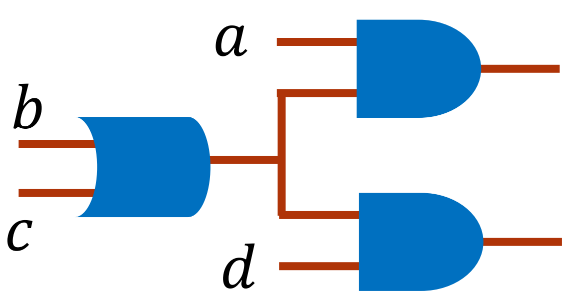
Library



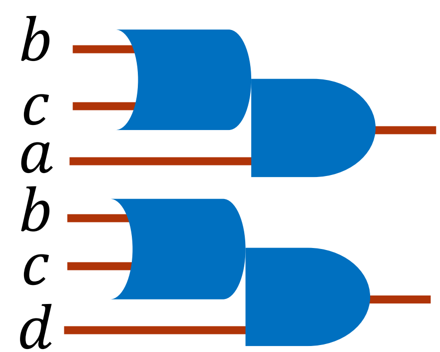
Target



Obvious Mapping



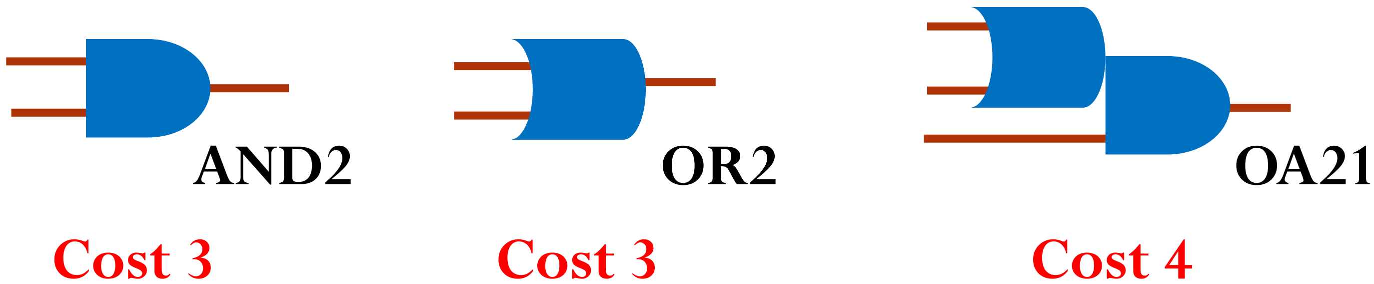
Non-obvious Mapping



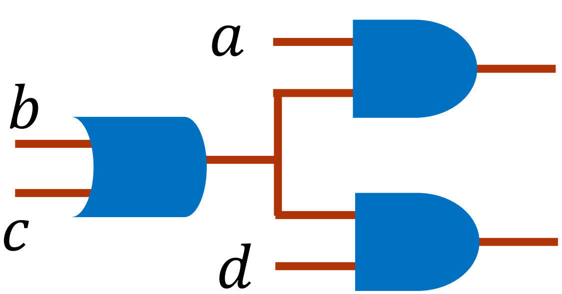
Tech Mapping

- Why choose a **non-obvious** mapping?
  - <u>**Answer**</u>: **Cost**. Suppose each gate in library has a cost associated with it, e.g., the **silicon area** of the standard cell gate.

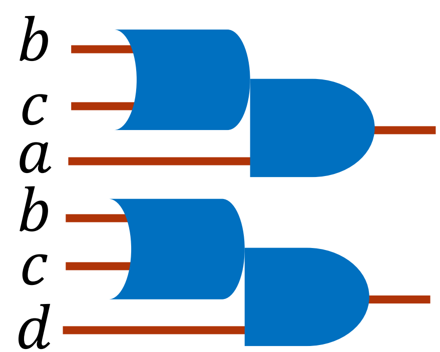
Library



Obvious Mapping	**Non-obvious Mapping**



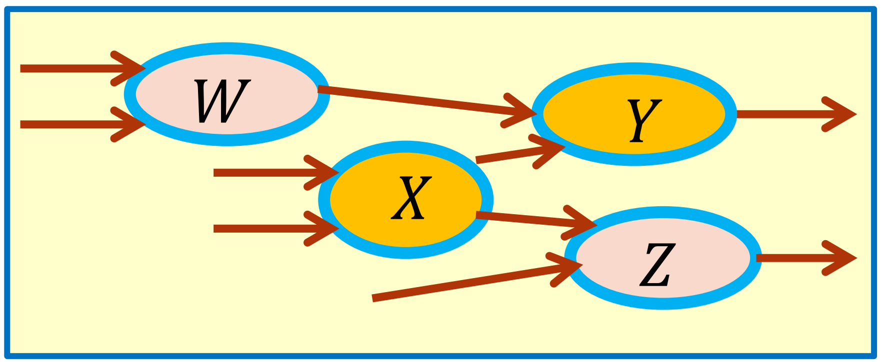
Cost 9

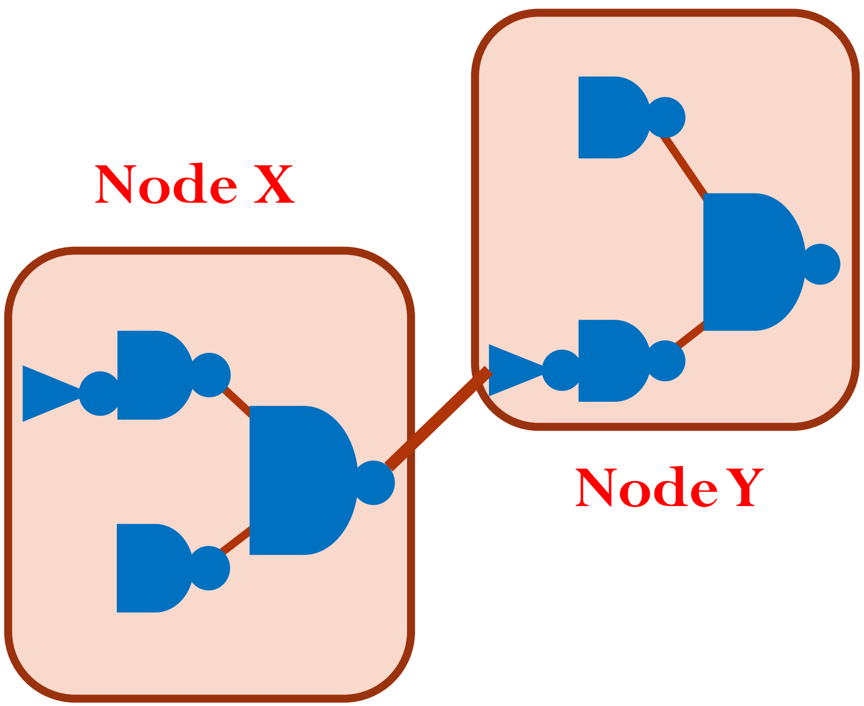


Cost 8

The Starting Point of Tech Mapping

- First, we transform **uncommitted logic** into simple, **real** gates.
  - We transform **every** SOP form in **each** node into **2-input NAND (NAND2)** & **NOT** gates. Nothing else!





How to Map SOP into NAND2s & NOTs?

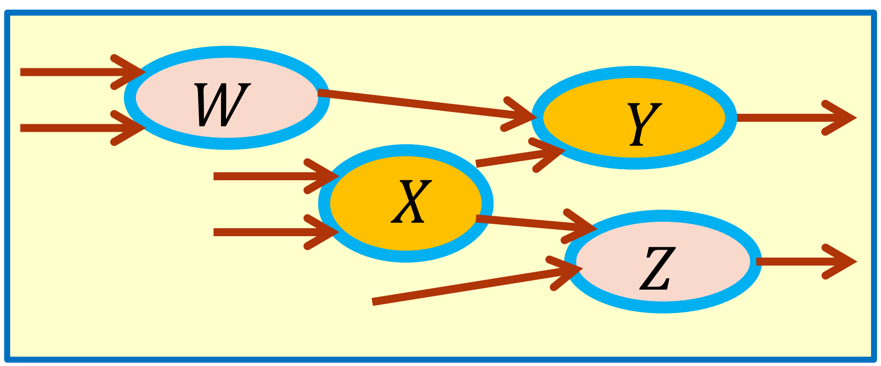
- Multi-input AND gate to multiple 2-input AND gates
- Multi-input OR gate to multiple 2-input OR gates
- 2-input AND gate

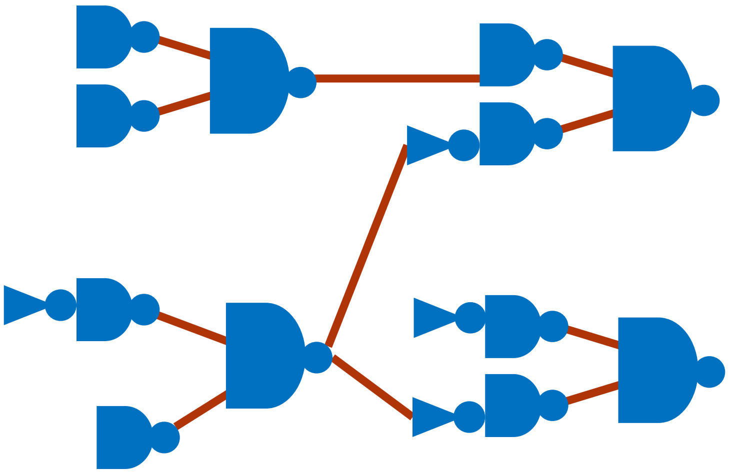
- 2-input OR gate

By De Morgan's Law

The Starting Point of Tech Mapping

- By transforming every SOP form in each node into NAND & NOT gates …
  - … Boolean logic network **disappears**.
  - We have one BIG "**flat**" network of **NAND2s** and **NOTs**. This is what we are going to map.





Outline

- Technology mapping
  - Problem
  - Background on the Solution -- Tree Covering Algorithm
  - Details of Tree Covering Algorithm

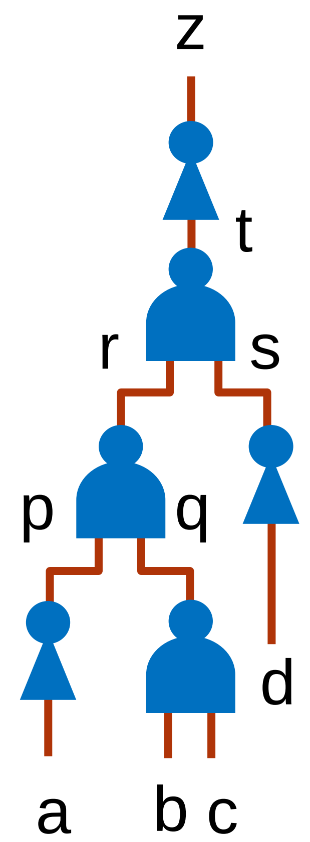
Technology Mapping as Tree Covering

- One famous, simple model of problem:
  - Your logic network to be mapped is **a tree of simple gates**.
    - We assume uncommitted form has **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.

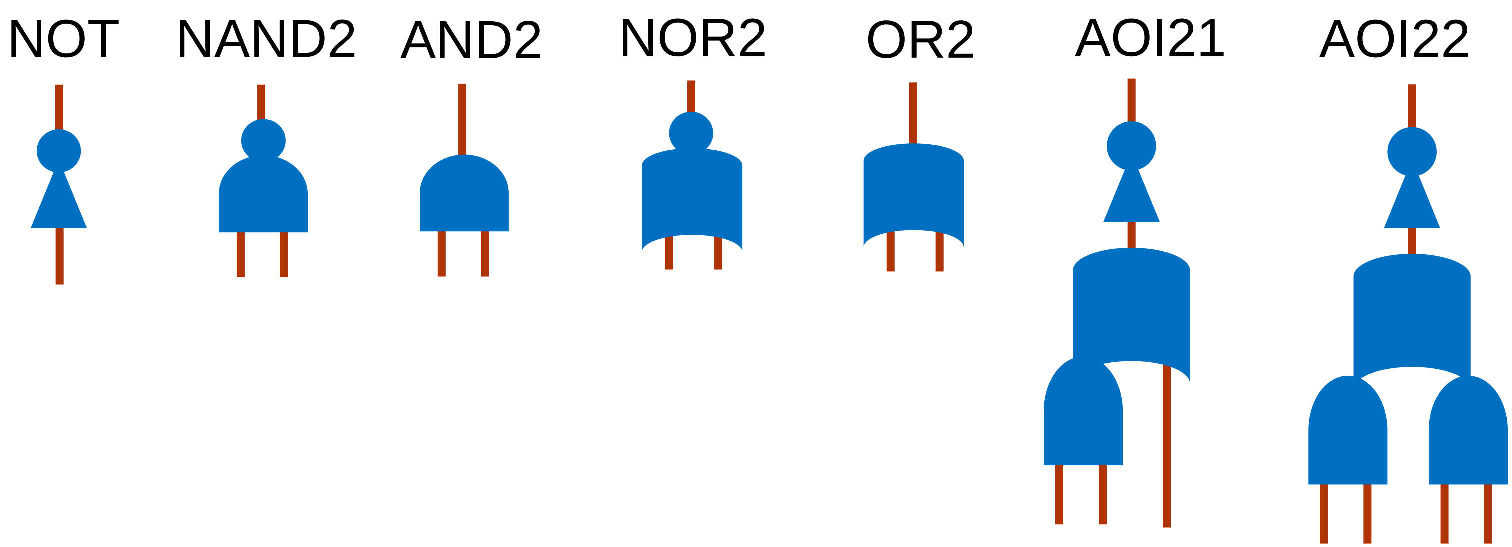
Tree Covering Example: Start with…

- Here is your uncommitted logic to be **tech mapped**.
  - This is the result from our technology independent synthesis, after replacing all SOP forms in the network nodes with **NAND2/NOT**.
  - Called the **subject tree**.
  - Label not only inputs but also all internal wires too.



Tree Covering: Your Technology Library

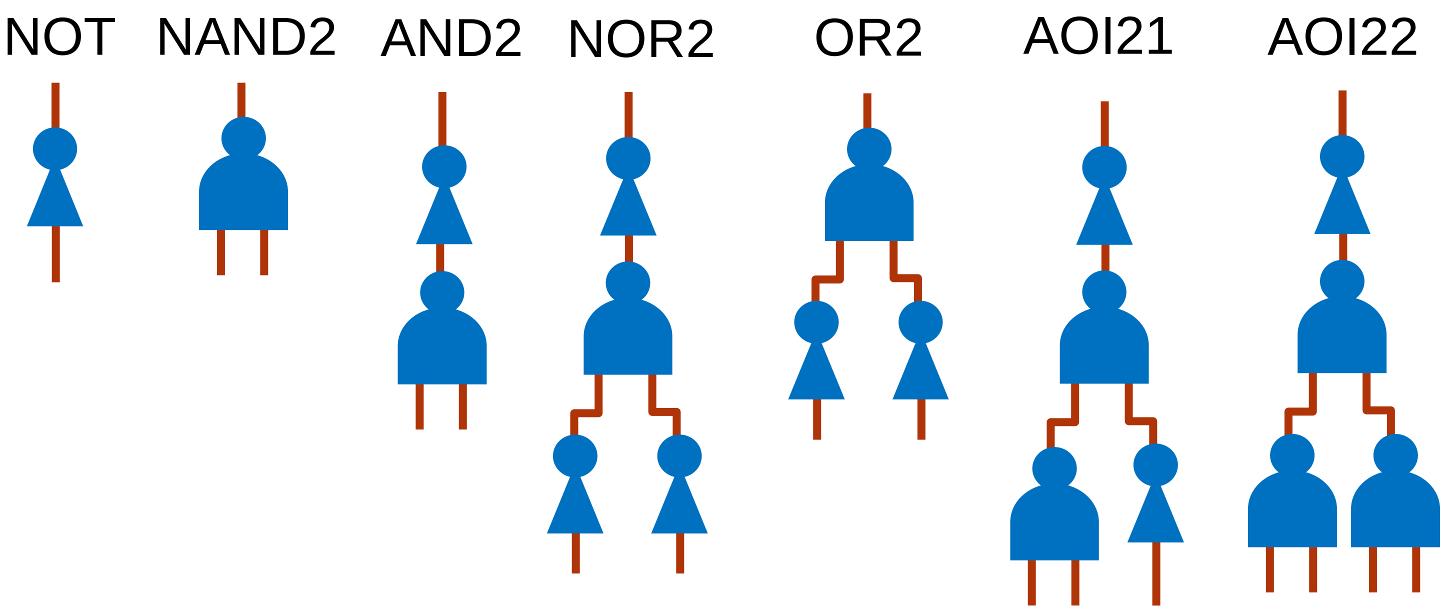
- And, here is a very simple **technology library**.



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

Tree Covering: Representing Library

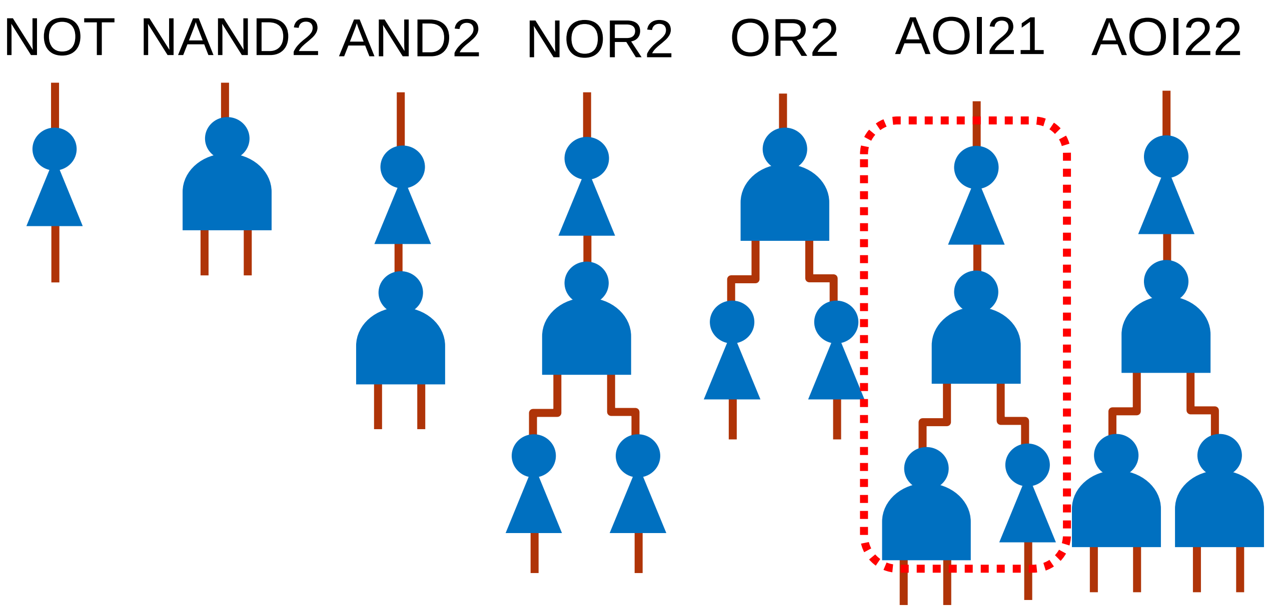
- Transforming to NAND2/NOT form is **easy**.
  - Just apply **De Morgan's law**.

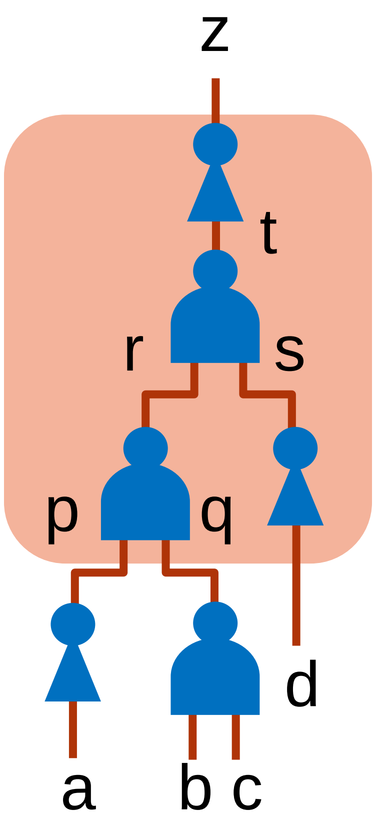


- Each library element in this form is called a **pattern tree**.

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**.



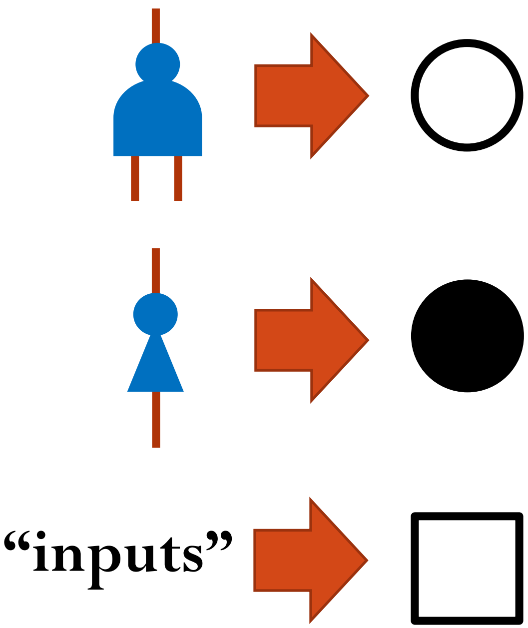


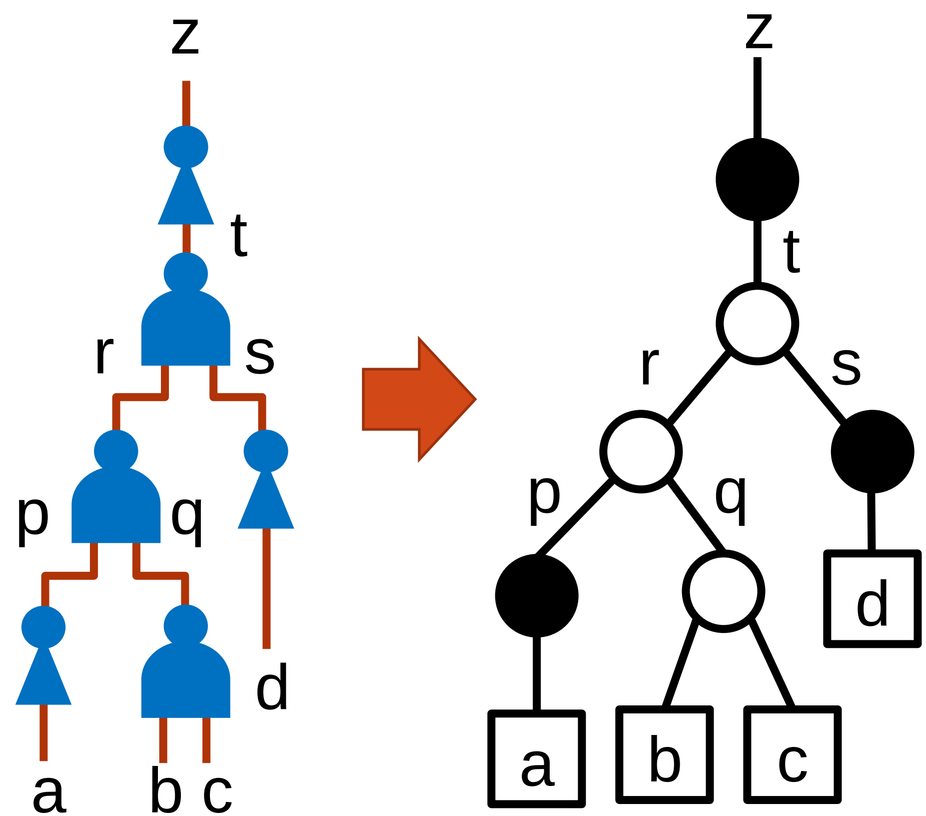
The General "Tree Covering"

- This is called a **structural tech mapper**.
- Why?
  - Because there is no Boolean algebra here!
  - We just match the gates and 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.

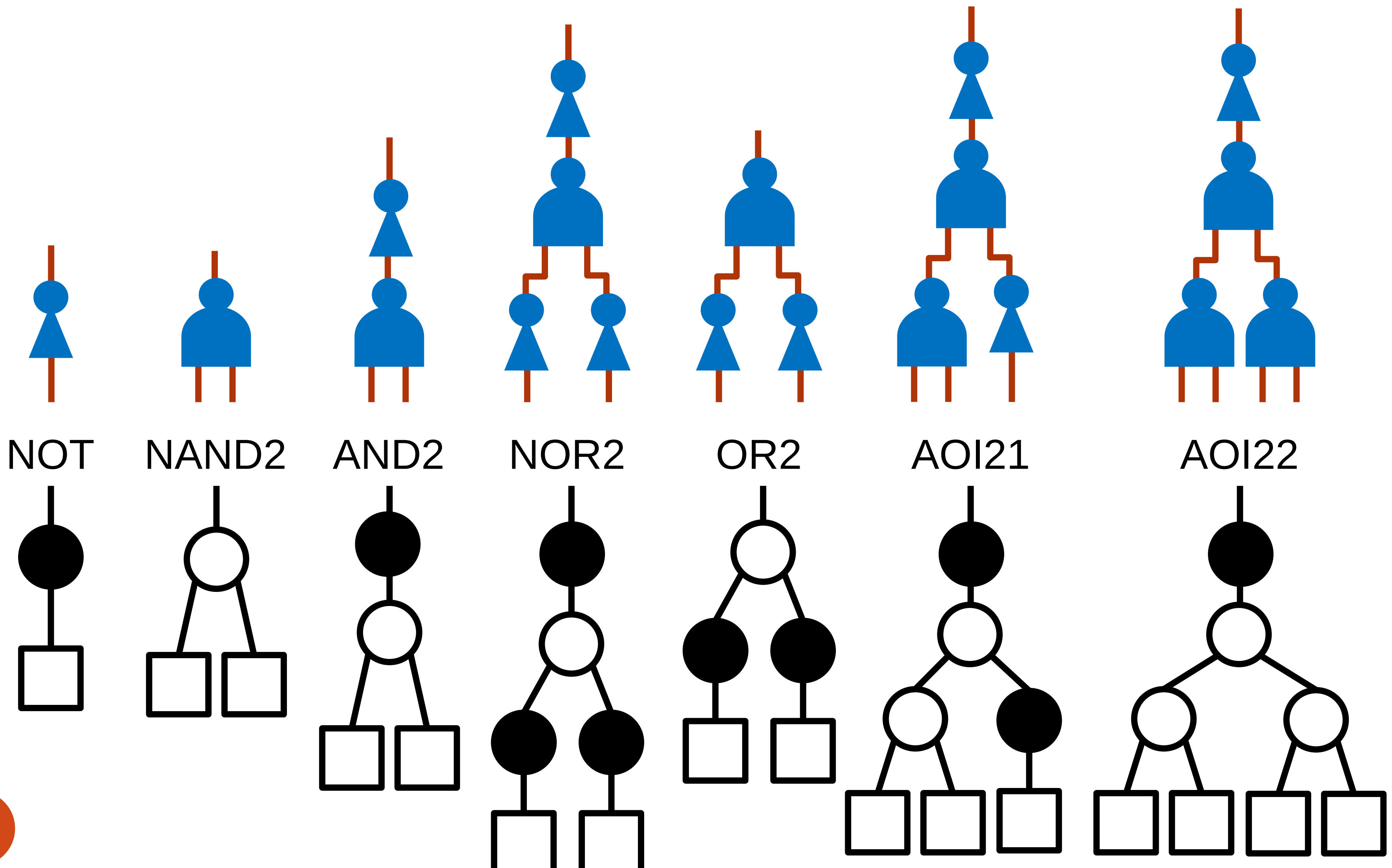
Representing Trees for Covering

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

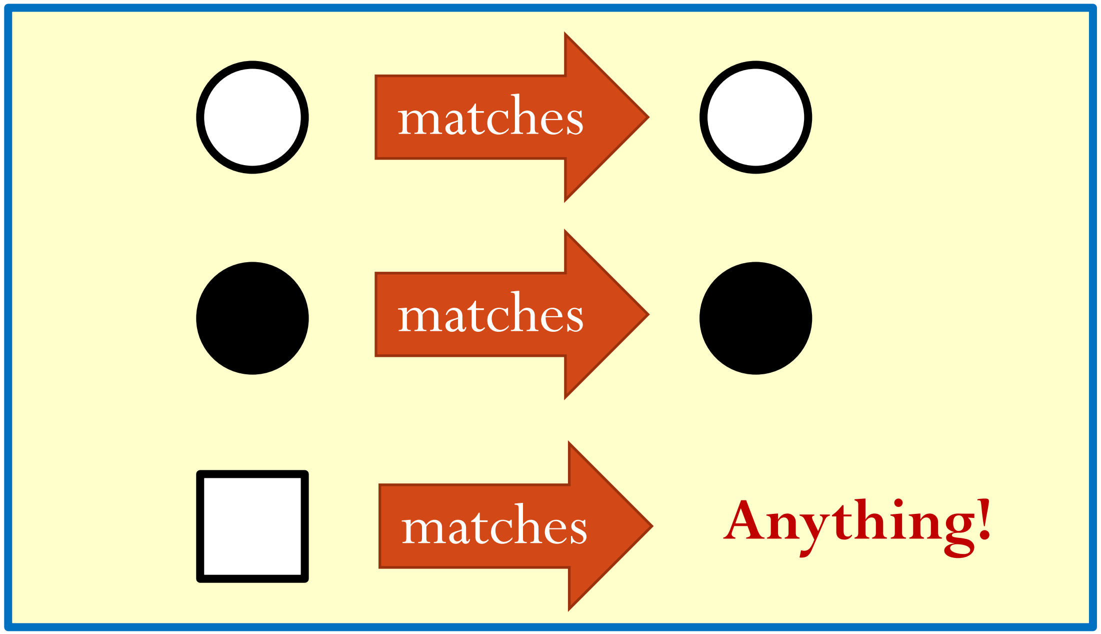


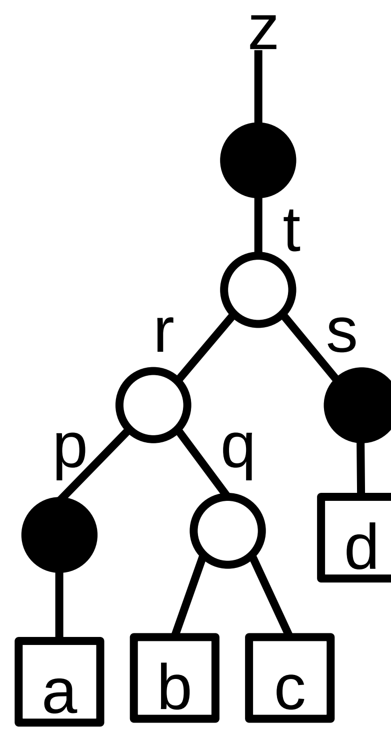


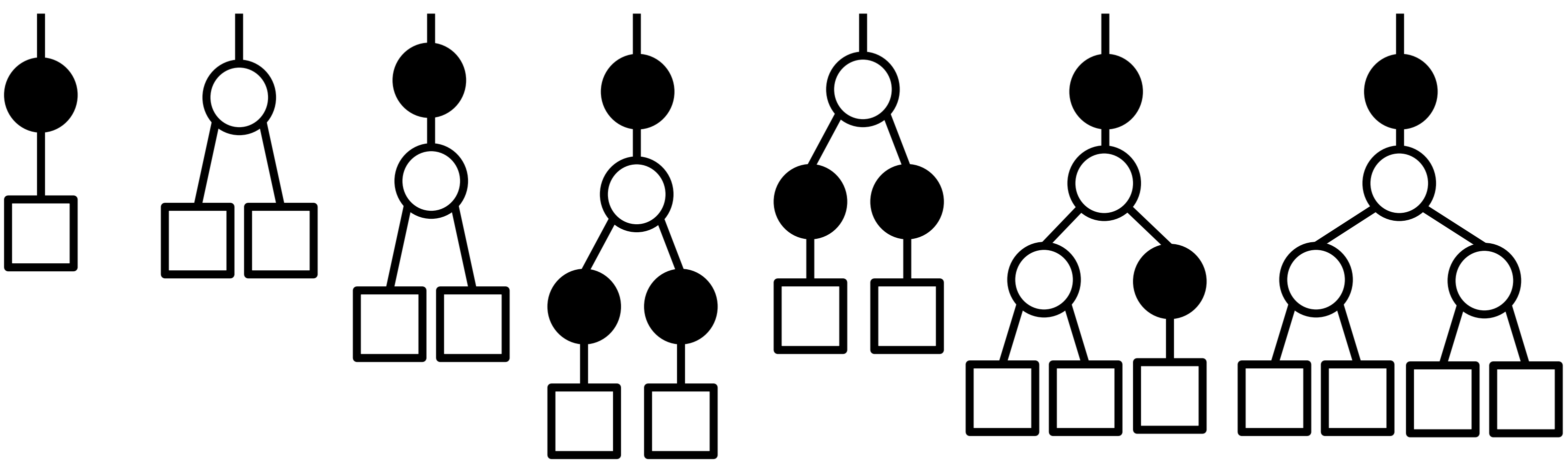
Represent Library in this Same Style



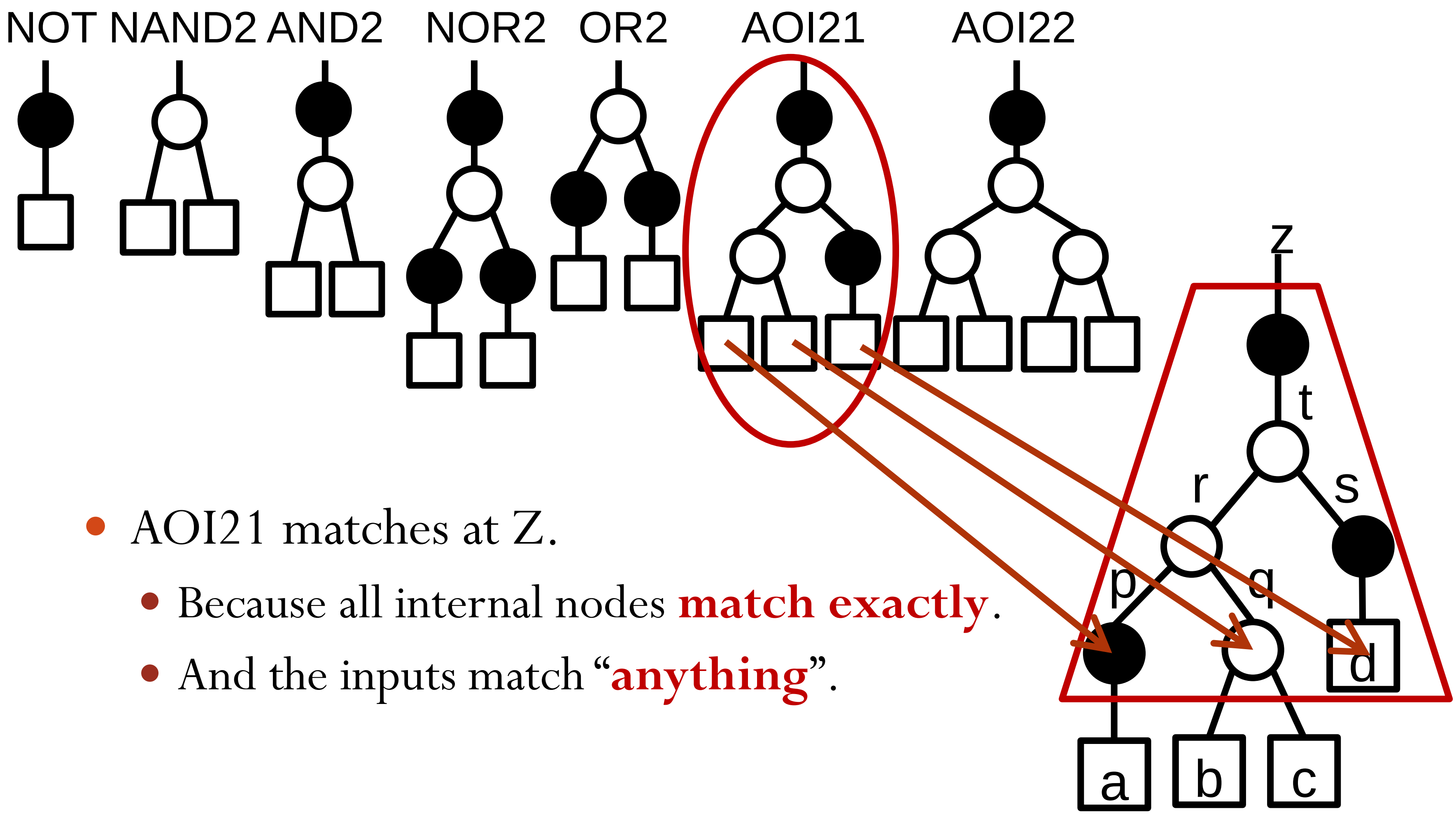
Structural Mapping Rules







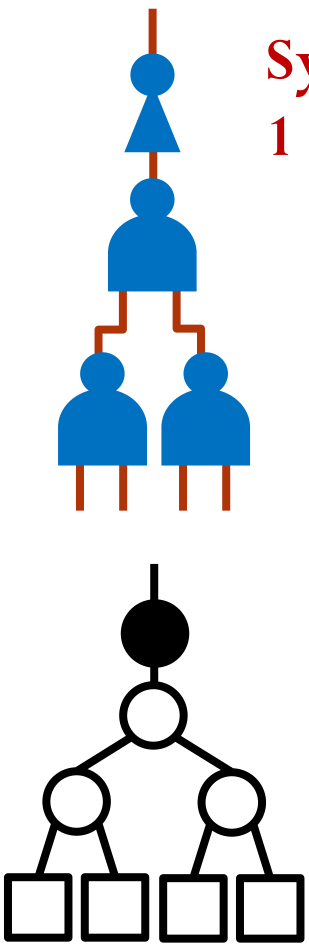
How a "Target Gate" Matches Subject Tree



- AOI21 matches at Z.
  - Because all internal nodes **match exactly**.
  - And the inputs match "**anything**".

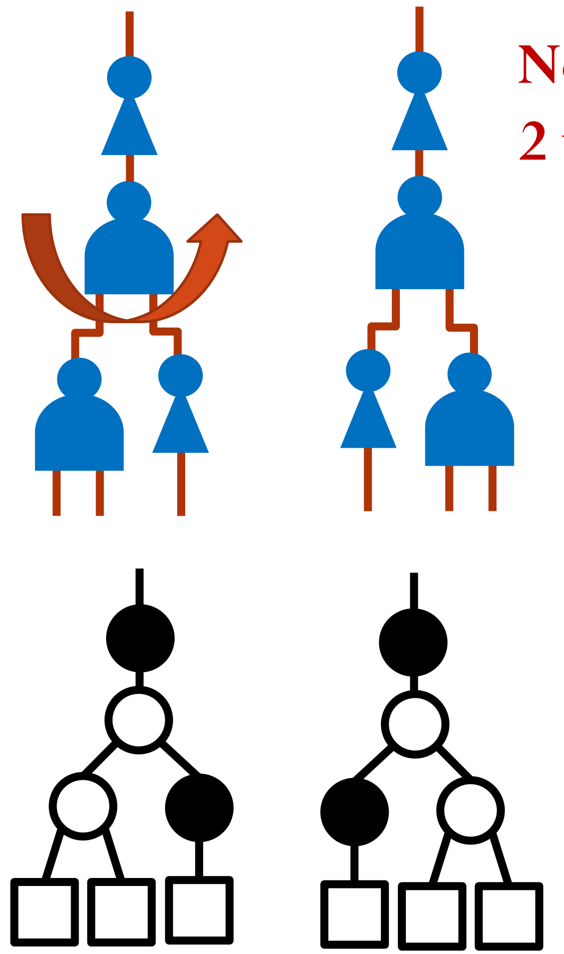
Be Careful: Symmetries Matter!

AOI22



Symmetric tree,
1 way to match

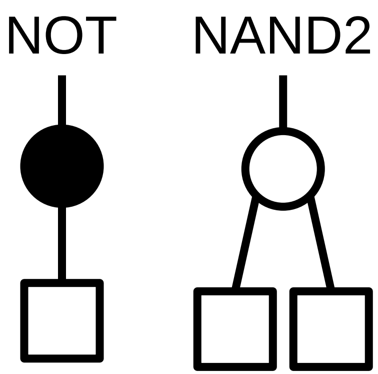
AOI21



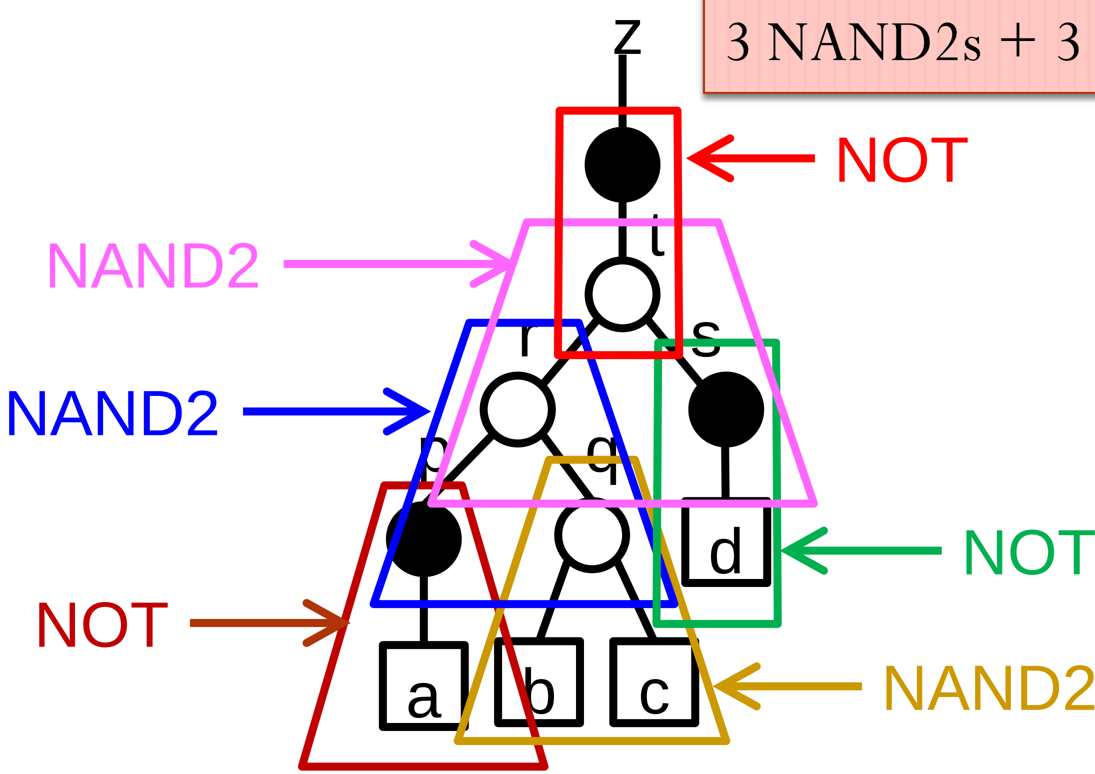
Not symmetric,
2 ways to match

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.

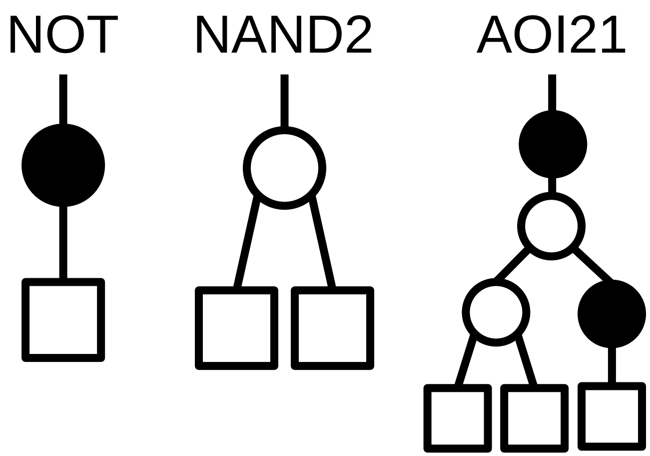


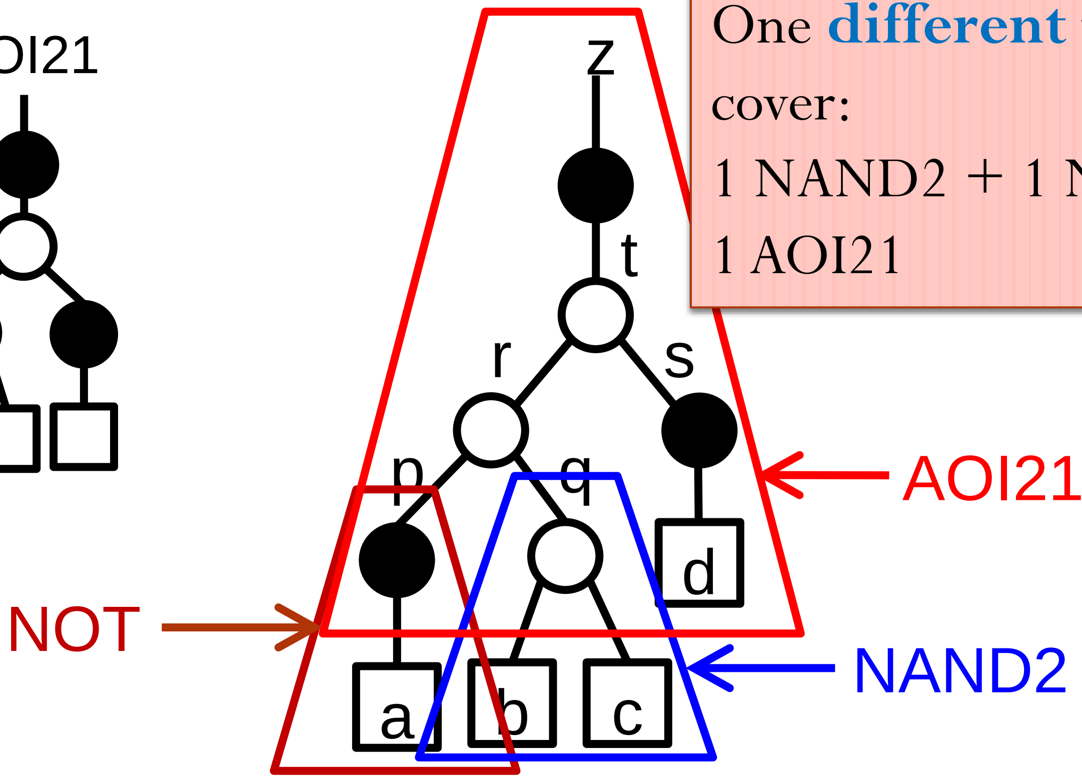
One correct tree cover:
3 NAND2s + 3 NOTs



Rules for a Complete Tree Cover

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





One **different** tree cover:

1 NAND2 + 1 NOT + 1 AOI21

Outline

- Technology mapping
  - Problem
  - Background on the Solution -- Tree Covering Algorithm
  - Details of Tree Covering Algorithm

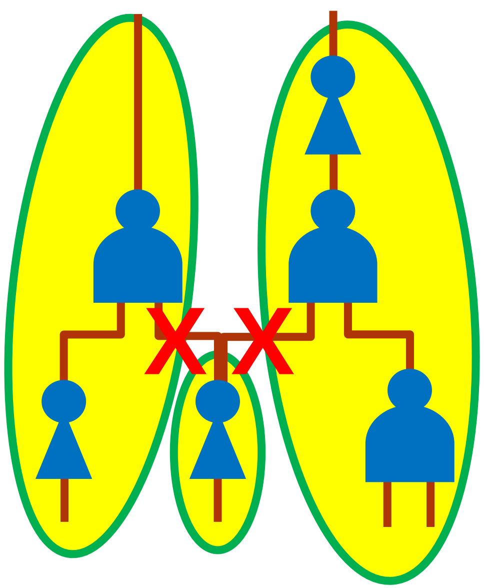
Tech Mapping via Tree Covering

- 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?

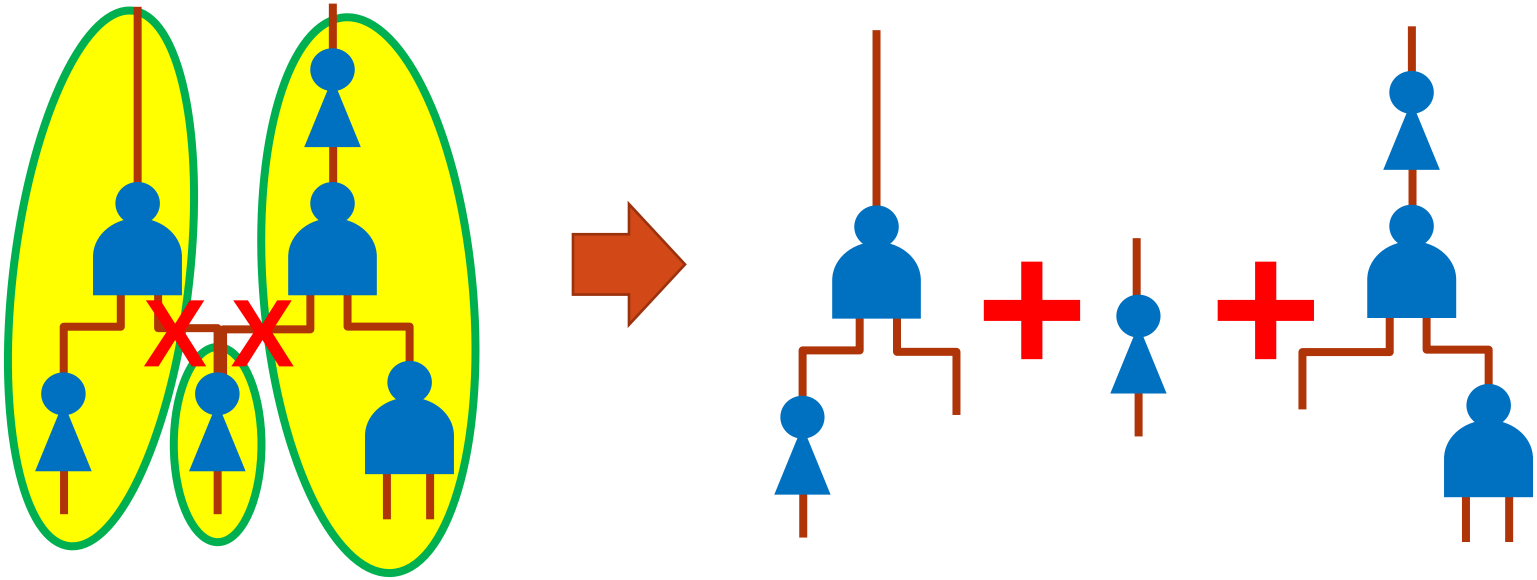
Treeifying the Netlist

- 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.

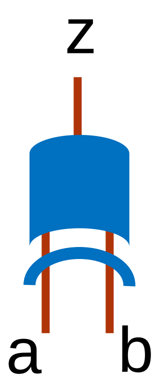
Treeifying Netlist: Result



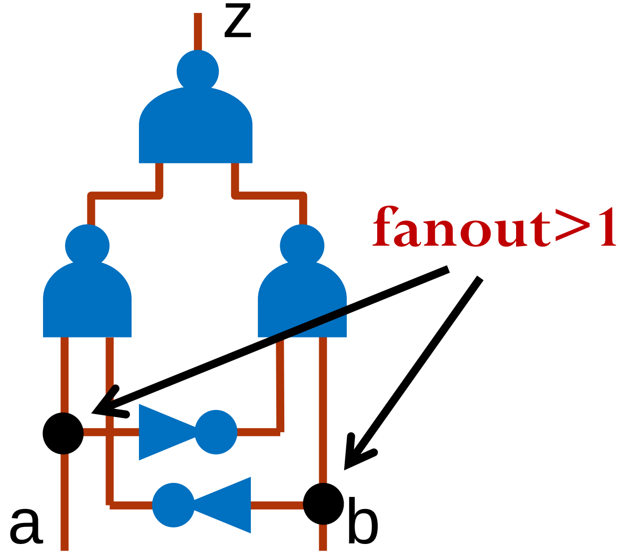
- We’re going to map these **3 trees** separately.
- This entails some loss of optimality, since **cannot** map across trees.
  - There are ways around these, but we won’t discuss these.

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!



$z = a\bar{b} + \bar{a}b$



So, no XOR gates for mapping!

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?

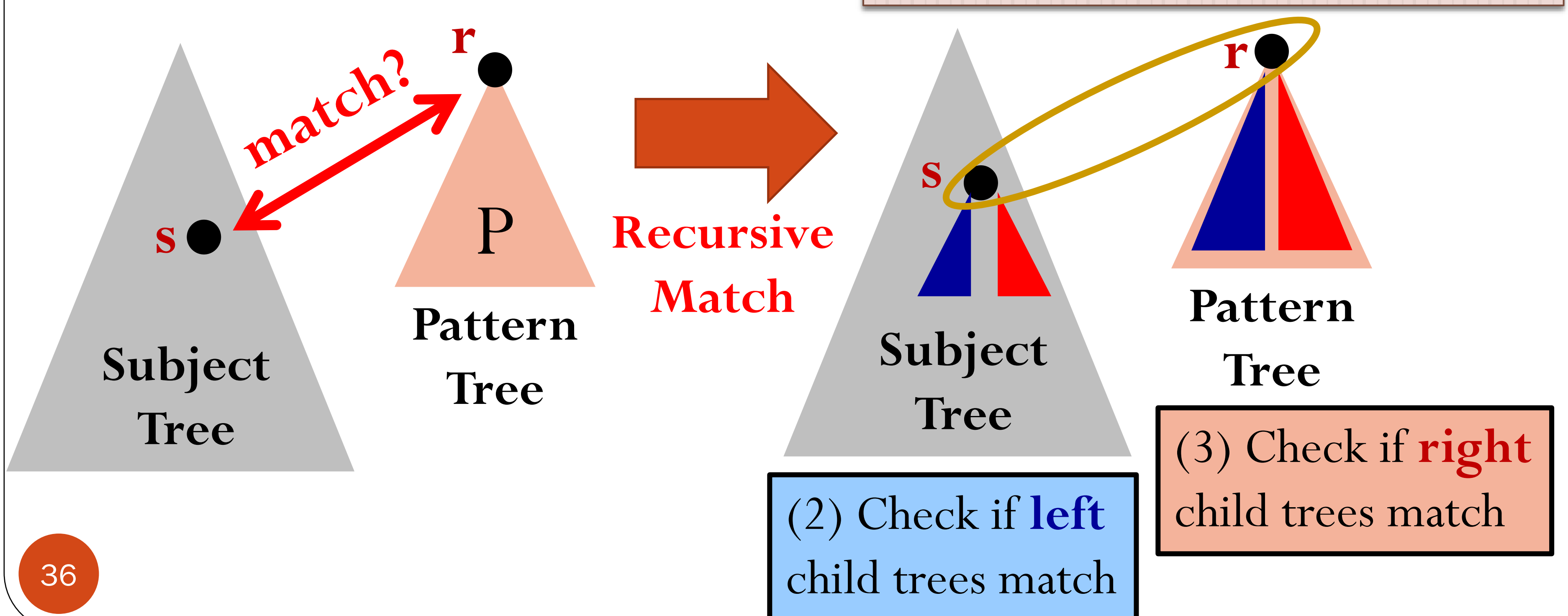
Tree Matching

- **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 a node n of subject tree with **root** of pattern tree, and then **recursively match children** of n in subject tree to **children** of pattern tree.

Recursive Tree Matching

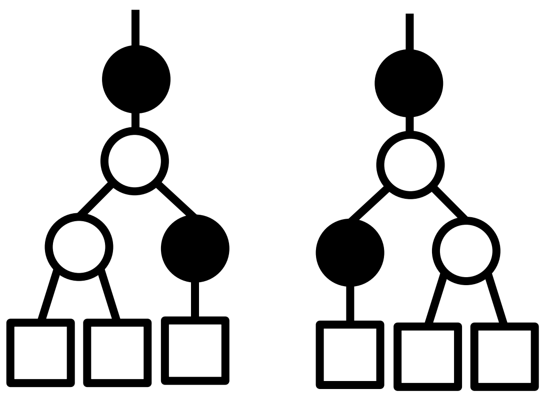
- Does library pattern tree P match node s in our subject tree?
  - First, check if node s matches root r of pattern P.
  - If so, **recursively** match **left child trees** of s and r, and then **right child trees** of s and r.



Tree Matching: One Subtlety

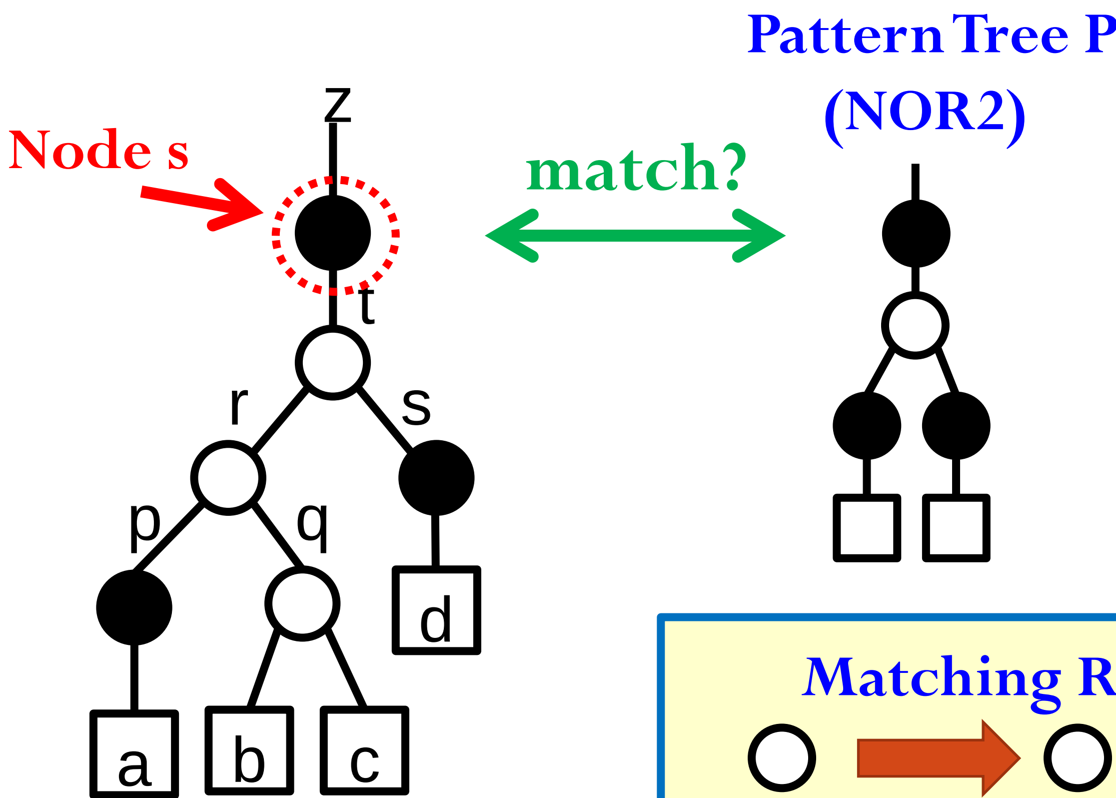
- Be careful matching asymmetric library patterns.
  - One example was AOI21. Need to check all possible matches by "**rotating**" the pattern tree.

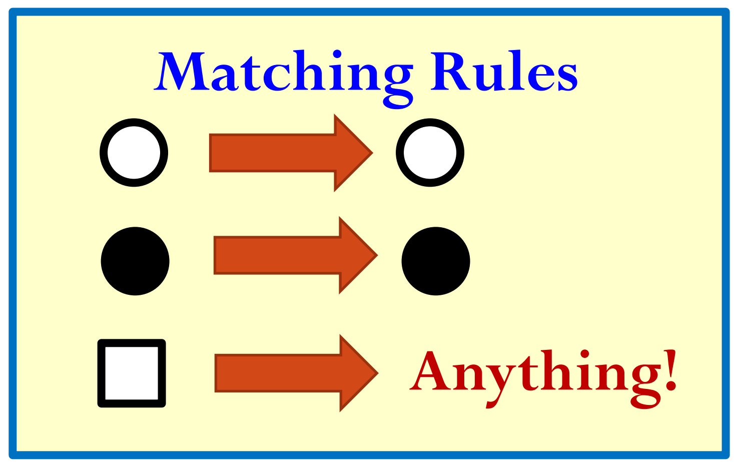
AOI21



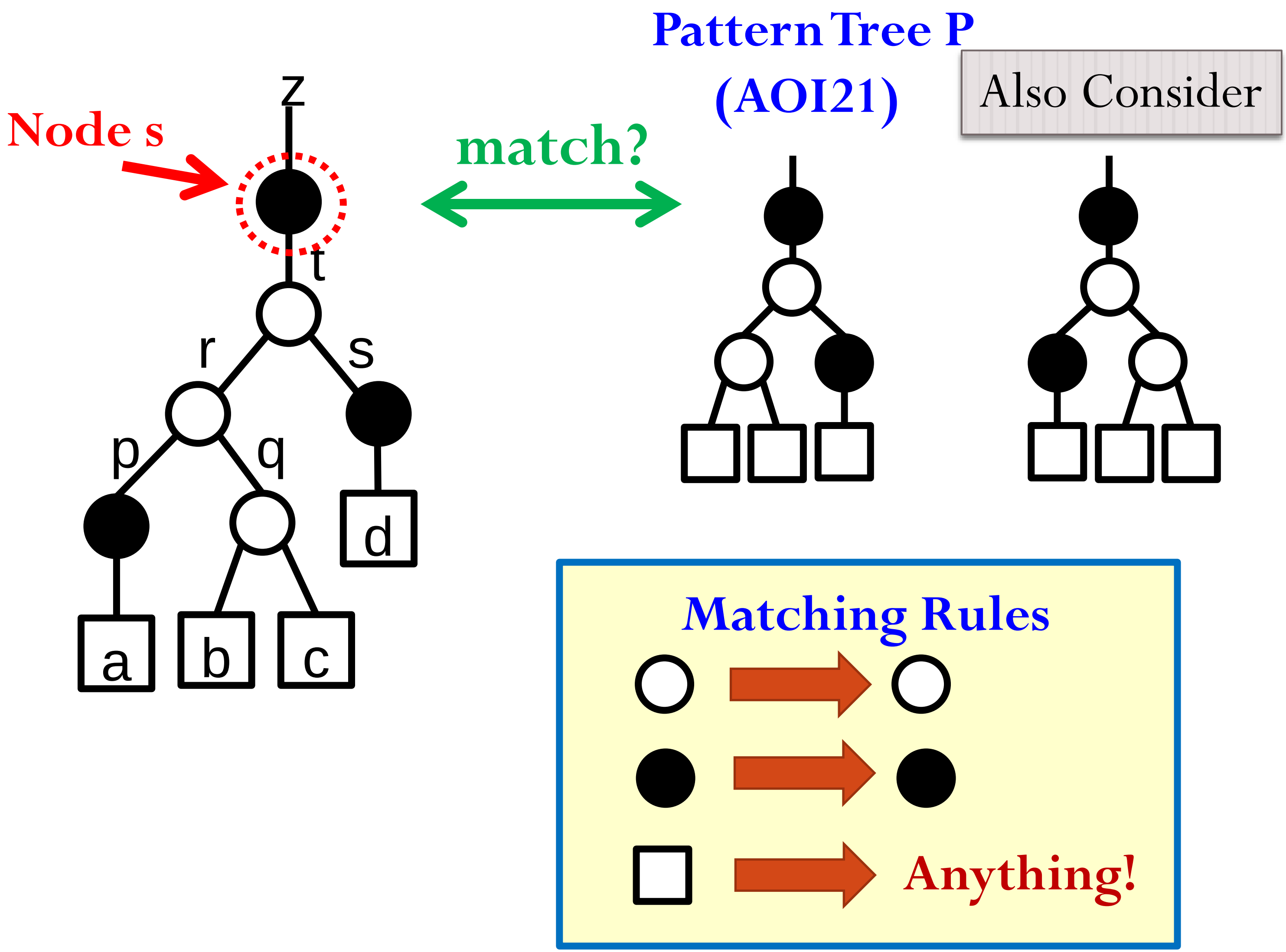
Both orientations are possible!

Tree Matching Example





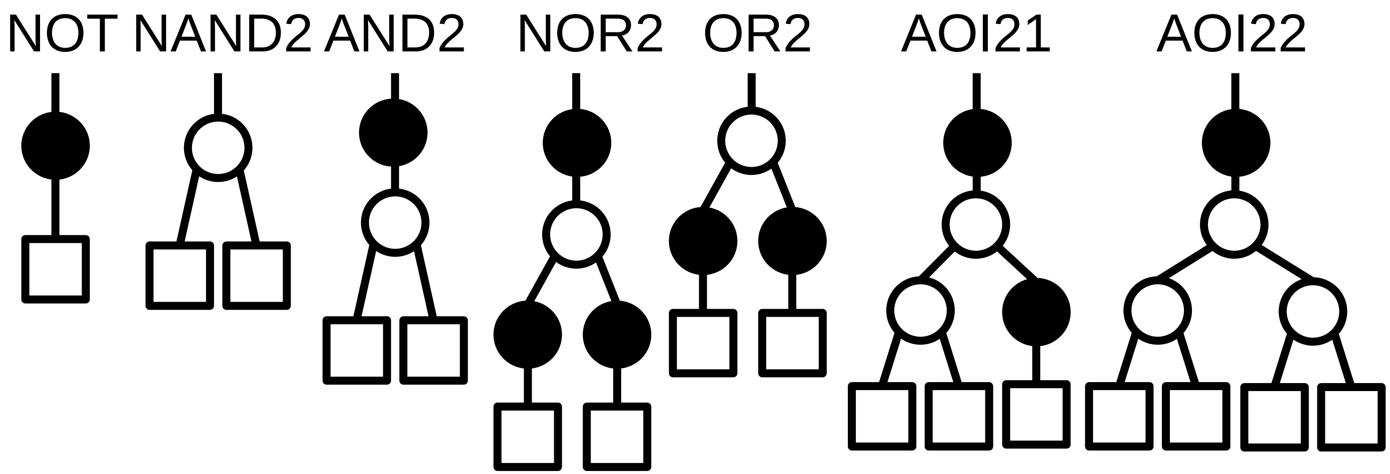
Tree Matching Example

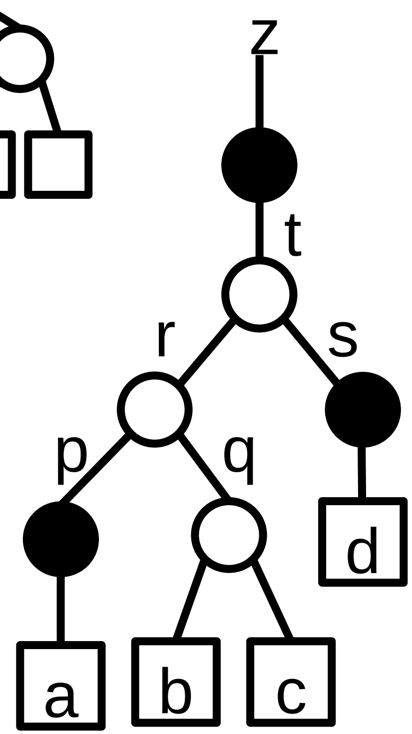


Result After Matching

- 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

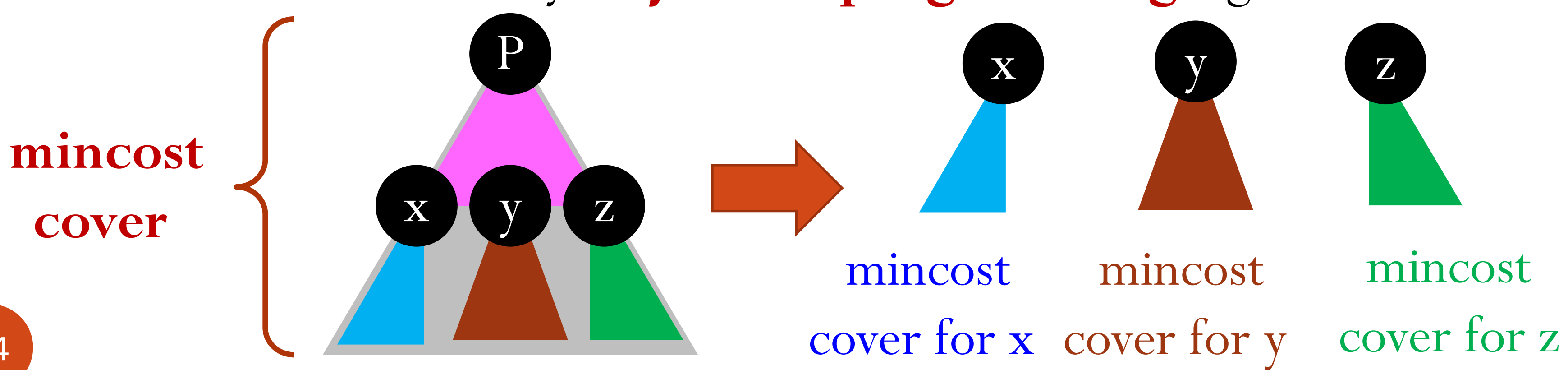
- **Subroutines**:
  - **Treeifying** the input netlist
    - Loses some optimality, but make things simple.
  - **Tree matching**
    - For each node in the subject tree, find pattern trees in library that **match**.
  - **Minimum-cost covering**
    - Assume you know what can match at each node of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

Minimum Cost Covering of Subject Tree

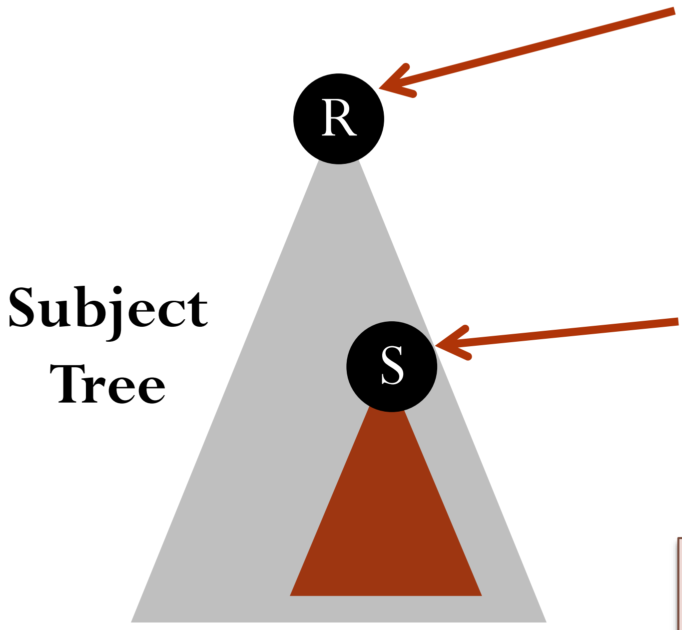
- What cover do we choose?
  - We assign a **cost** to each library pattern.
  - We choose a **minimum cost** ("**mincost**") cover of the subject tree.

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



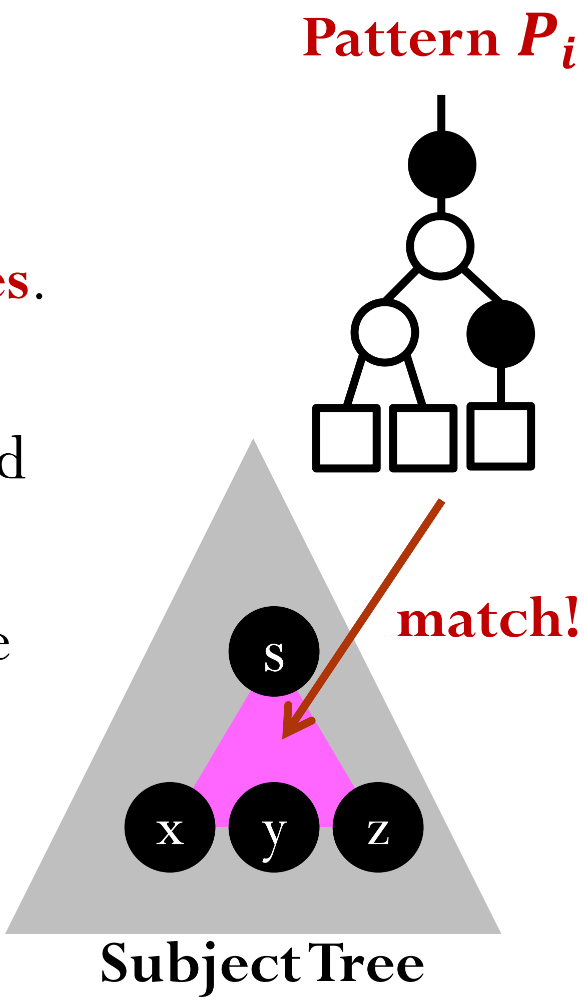
Root R of subject tree. Best cover (mapping) of this tree has cost = **mincost(R)**

Internal node s of subject tree. Best cover (mapping) of this tree has cost = **mincost(s)**

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.

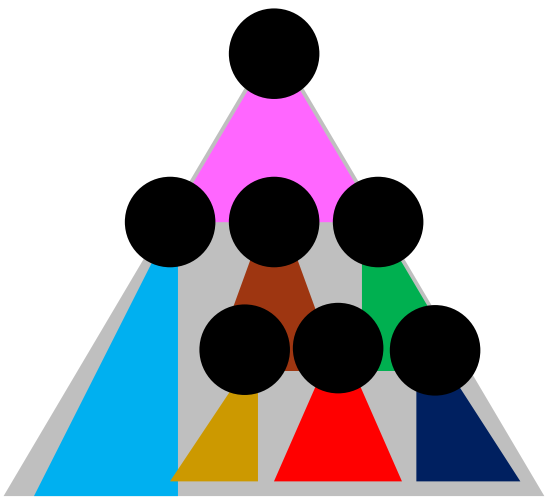
Some Terminology (cont.)

- **<u>Suppose</u>**:
  - 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

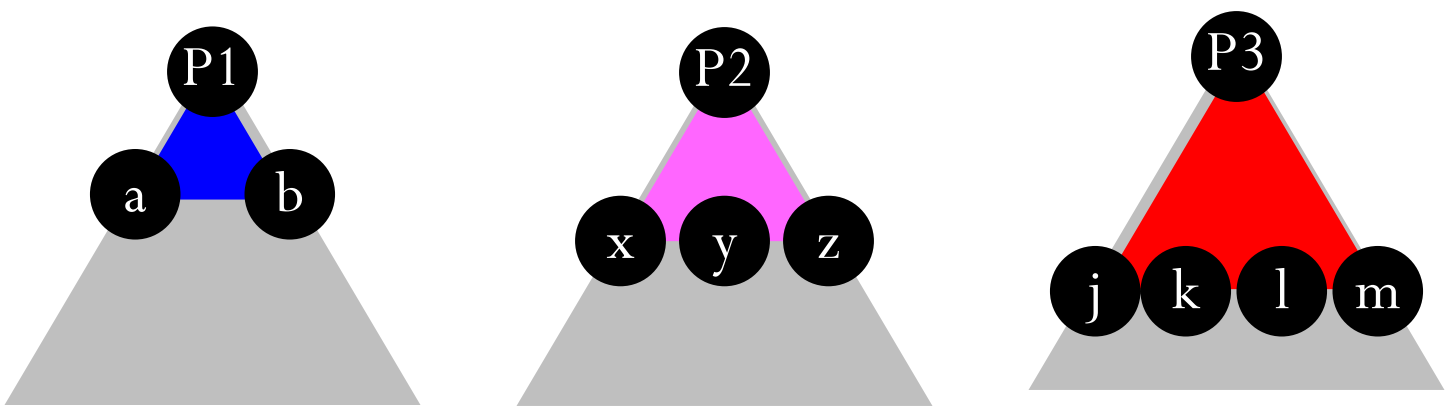
- Every gate pattern in target library has a **cost**.
  - Gate pattern P_i has $\text{cost}(P_i)$.
- To calculate cost of mapping the **entire** subject tree:
  - We add up **costs** of all pattern trees covering subject.



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

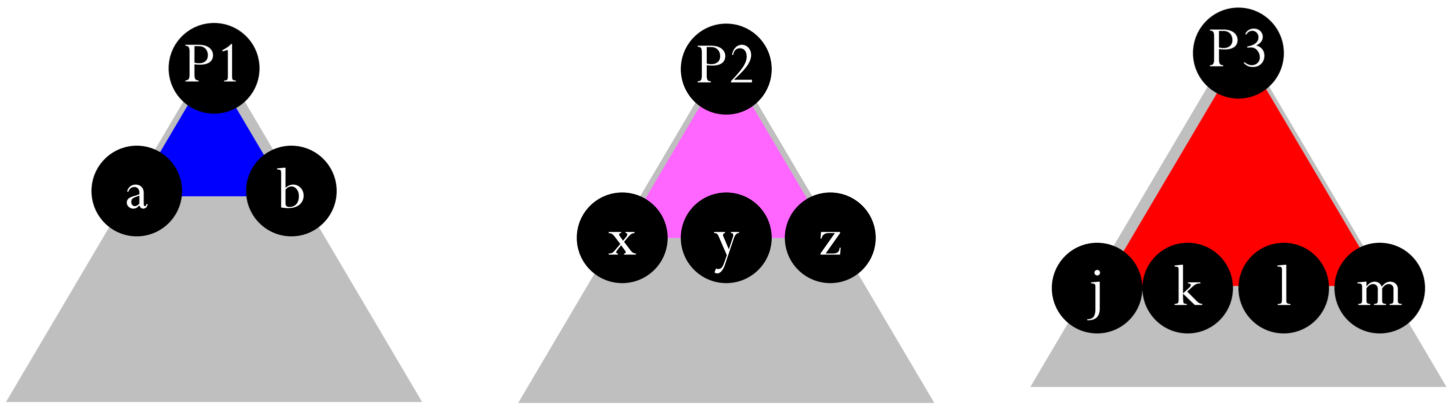
Mincost Tree Covering: The Idea

- 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 produce the **smallest** value of 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 **}**

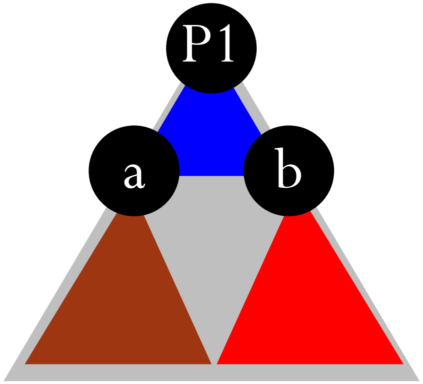
What're these?

Mincost Tree Cover: Recursive Formula

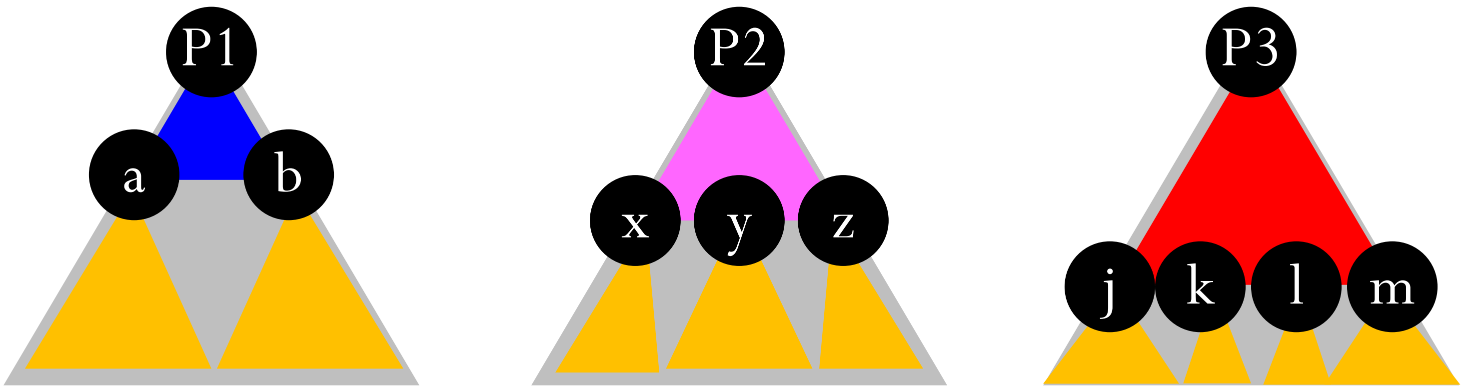
- In calculating **mincost**(R), we need to answer:
  - What is **minimal** cost with **P1** matching root R?

- **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

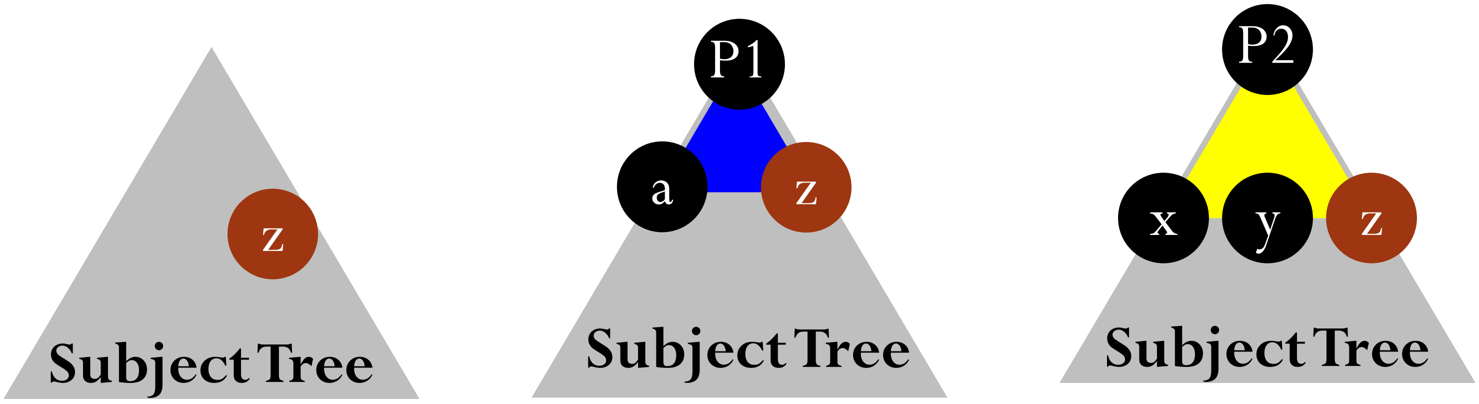
```
mincost( treenode) {
    cost = ∞
    foreach( pattern P matching at subject treenode) {
        let L = {nodes in subject tree corresponding to leaf nodes in P
            when P is placed with its root at treenode }
        newcost = cost(P)
        foreach( node n in L } {
            newcost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

Min Cost Tree Cover

- There is **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.

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.



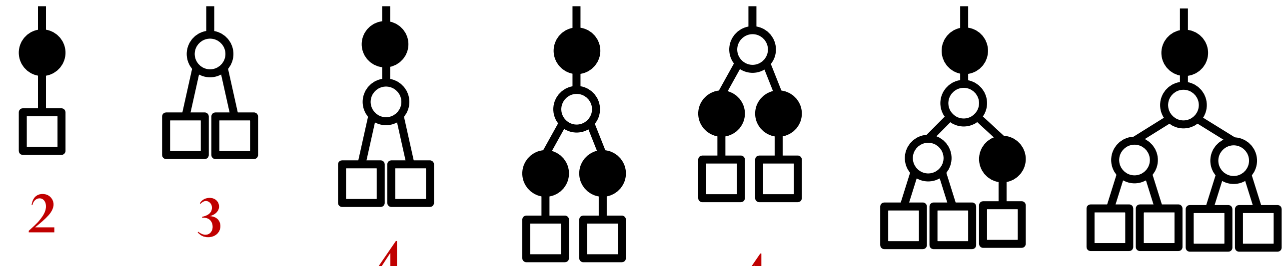
Better Solution

- **Basic idea**: just compute it **once**. First time, **save** it; next time, **look it up**!

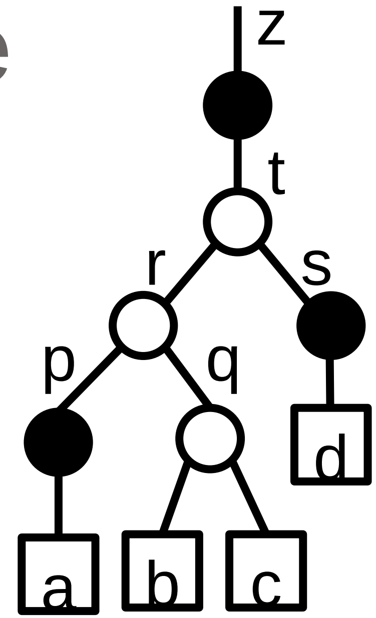
- **Details**:
  - Keep a table with **mincost** value for each node
  - Start each entry with value ∞
  - Each time computing **mincost(node), check <u>first</u>** to see if **node** has been computed (i.e., whether the value is ∞)
    - No: compute it and save the value in the entry
    - Yes: simply read the value

Min Cost Tree Cover Example

	NOT	NAND2	AND2	NOR2	OR2	AOI21	AOI22



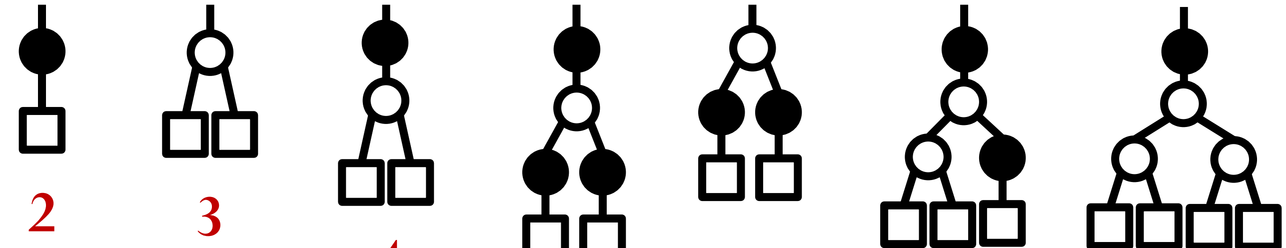
Cost	**2**	**3**	**4**	**6**	**4**	**7**	**7**



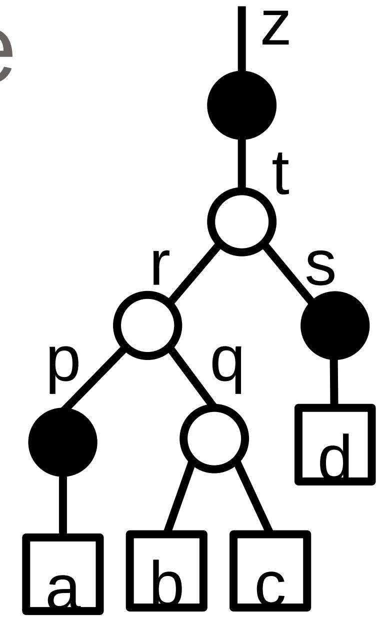
Node	**Can Match**	**Mincost**
z	NOT	2+mincost(t)
	AND2	4+mincost(r)+mincost(s)
	AOI21	7+mincost(p)+mincost(q)
		min
t	NAND2	3+mincost(r)+mincost(s)
r	NAND2	3+mincost(p)+mincost(q)
p	NOT	2 ⇒ mincost(p)=2
q	NAND2	3 ⇒ mincost(q)=3
s	NOT	2 ⇒ mincost(s)=2

Min Cost Tree Cover Example

NOT NAND2 AND2 NOR2 OR2 AOI21 AOI22

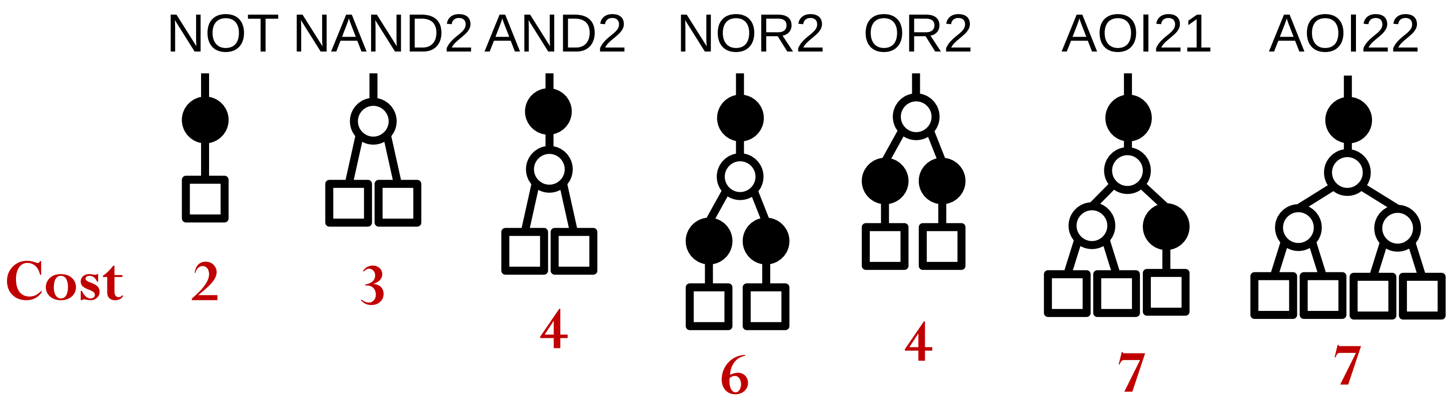


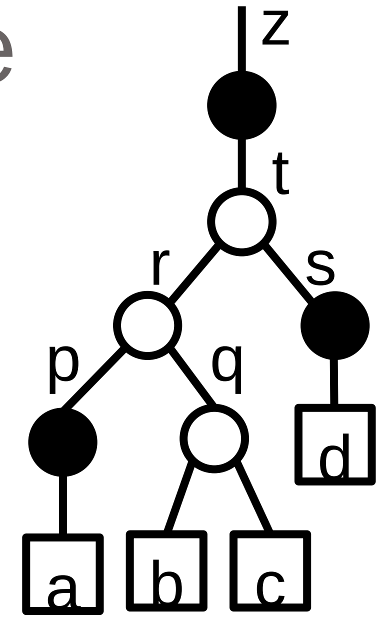
Cost 2 3 4 6 4 7 7



Node	Can Match	Mincost
z	NOT AND2 AOI21	min { 2+mincost(t) 4+mincost(r)+mincost(s) 7+mincost(p)+mincost(q) } ⇒ mincost=12
t	NAND2	3+mincost(r)+mincost(s)
r	NAND2	3+mincost(p)+mincost(q) ⇒ mincost(r)=8
p	NOT	2 ⇒ mincost(p)=2
q	NAND2	3 ⇒ mincost(q)=3
s	NOT	2 ⇒ mincost(s)=2

Min Cost Tree Cover Example

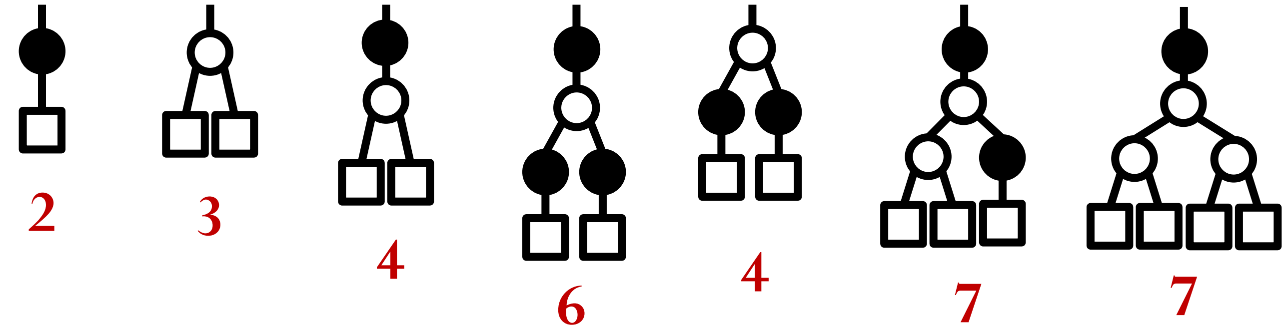




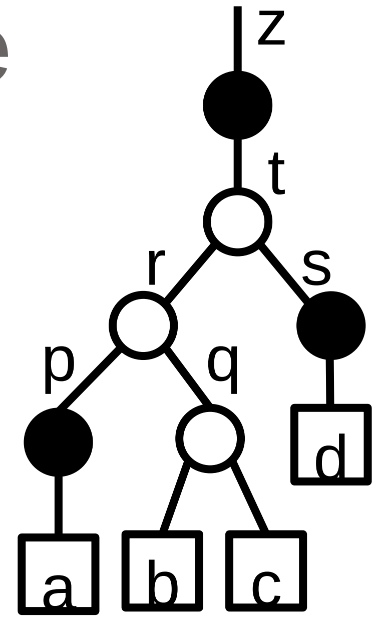
Node	Can Match	Mincost	
z	NOT AND2 AOI21	2+mincost(t) 4+mincost(r)+mincost(s) 7+mincost(p)+mincost(q)	min ⇒ mincost=14 ⇒ mincost=12
t	NAND2	3+mincost(r)+mincost(s)	⇒ mincost(t)=13
r	NAND2	3+mincost(p)+mincost(q)	⇒ mincost(r)=8
p	NOT	2 ⇒ mincost(p)=2	
q	NAND2	3 ⇒ mincost(q)=3	
s	NOT	2 ⇒ mincost(s)=2	

Min Cost Tree Cover Example

	NOT	NAND2	AND2	NOR2	OR2	AOI21	AOI22

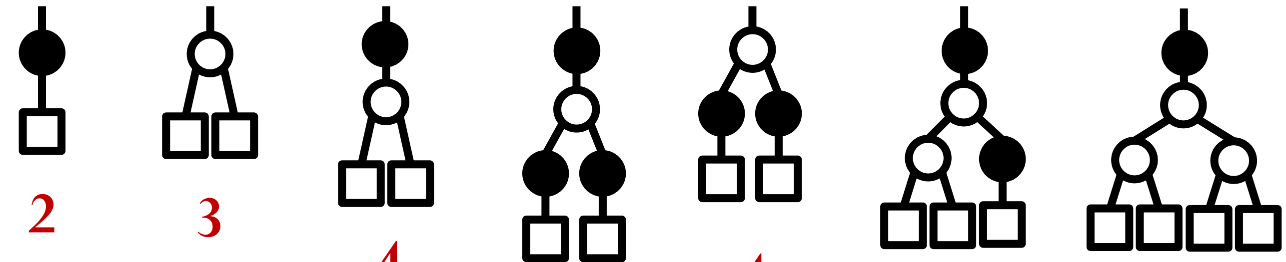


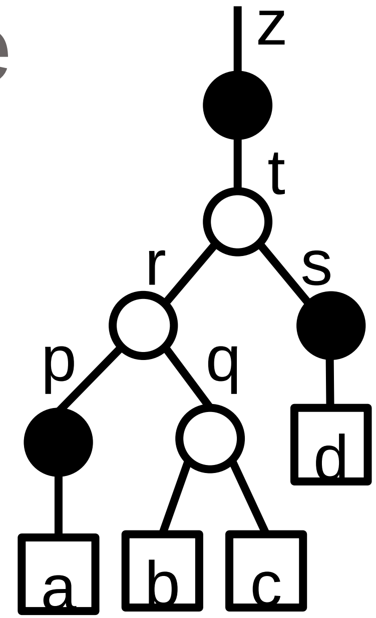
Cost	2	3	4	6	4	7	7



Node	**Can Match**	**Mincost**	
z	NOT	2+mincost(t)	⇒ mincost=15
	AND2	4+mincost(r)+mincost(s)	⇒ mincost=14
	AOI21	7+mincost(p)+mincost(q)	⇒ mincost=12
		min	
t	NAND2	3+mincost(r)+mincost(s)	⇒ mincost(t)=13
r	NAND2	3+mincost(p)+mincost(q)	⇒ mincost(r)=8
p	NOT	2	⇒ mincost(p)=2
q	NAND2	3	⇒ mincost(q)=3
s	NOT	2	⇒ mincost(s)=2

Min Cost Tree Cover Example

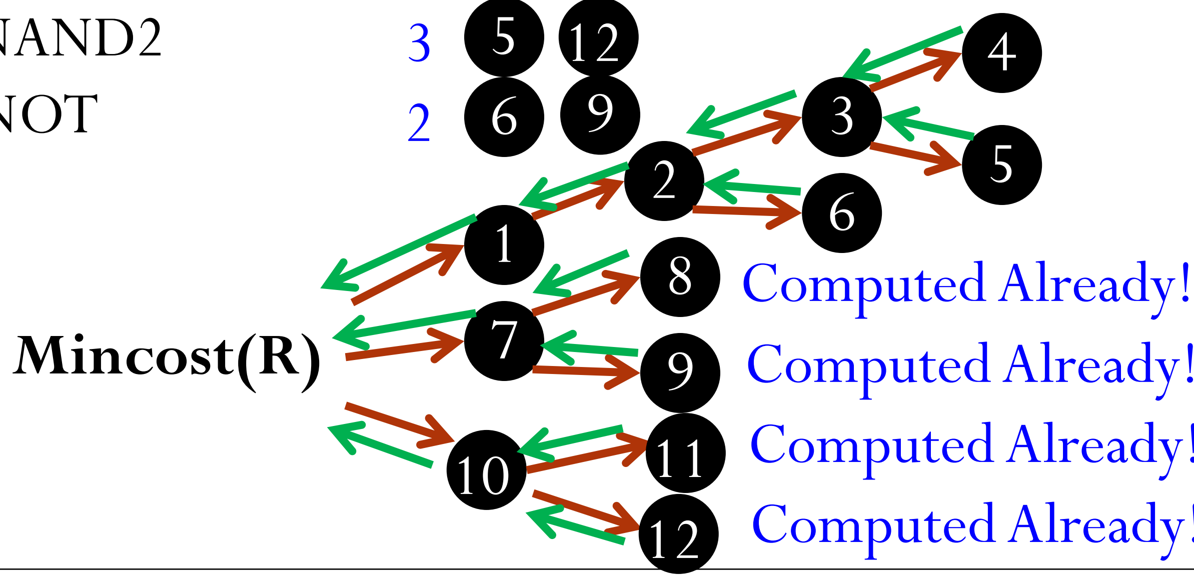




Node	Can Match	Mincost	
z	NOT	15	
	AND2	14	min
	AOI21	12	Best!
t	NAND2	13	
r	NAND2	8	
p	NOT	2	
q	NAND2	3	
s	NOT	2	

Actual Execution Sequence

Node	**Can Match**		**Mincost**
z	NOT AND2 AOI21	**min**	2+mincost(t) 1 4+mincost(r)+mincost(s) 7 7+mincost(p)+mincost(q) 10
t	NAND2		3+mincost(r)+mincost(s) 2
r	NAND2		3+mincost(p)+mincost(q) 3 8
p	NOT		2 4 11
q	NAND2		3 5 12
s	NOT		2 6 9

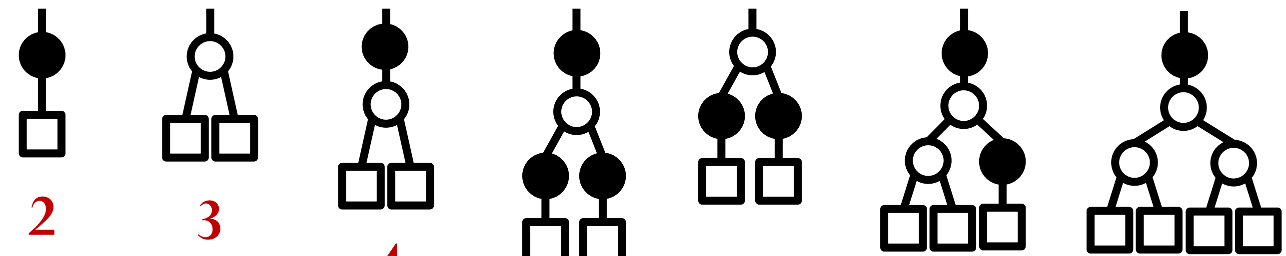


Min Cost Cover: How To Get Final Cover?

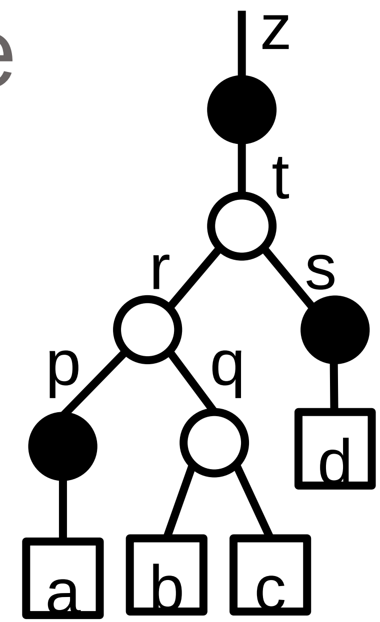
- Look at **best cost** at subject root. Find pattern P at the root that gives that cost.
- Find **leaf nodes** in the subject tree for pattern P.
  - Look at the **best cost** at each of these leaf nodes.
  - Find the pattern P_i that is associated with each of these best costs.
  - Look again at leaf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

Min Cost Tree Cover Example

	NOT	NAND2	AND2	NOR2	OR2	AOI21	AOI22



Cost	2	3	4	6	4	7	7



Node	**Can Match**	**Mincost**	
z	NOT	2+mincost(t) = 15	min
	AND2	4+mincost(r)+mincost(s) = 14	
	AOI21	7+mincost(p)+mincost(q) = 12	Best!
t	NAND2	3+mincost(r)+mincost(s) = 13	
r	NAND2	3+mincost(p)+mincost(q) = 8	
p	NOT	2	
q	NAND2	3	
s	NOT	2	

Min Cost Tree Cover

- Turns out to be several nice **extensions** possible
  - Can modify algorithm a little to minimize **delay** instead of area 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.
- 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 mapping for Lookup-Table (LUT) in FPGA.