Lecture 4: Logic Synthesis -2

ECE201A

Some notes adopted from
Andrew B. Kahng
Lei He
Igor Markov
Mani Srivastava
Synopsys
Various Sources

Logistics

- Students should help each other out on Piazza!
  - Please feel free to answer simple tool questions
- I hope you have started on Lab 1

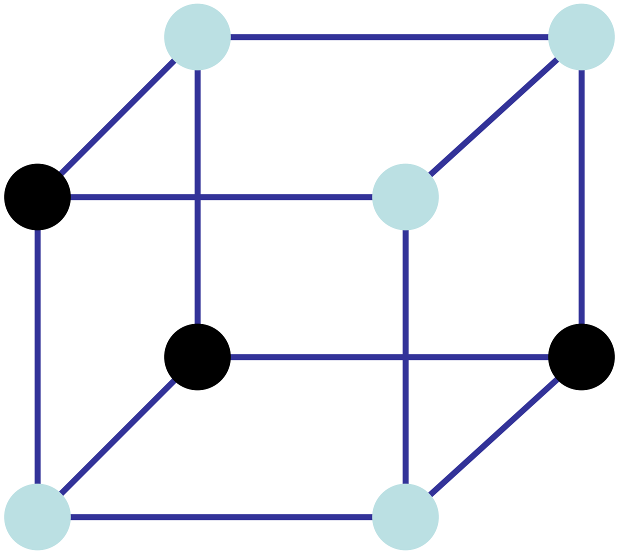
Quine-McCluskey Method

- We want to find a minimum prime and irredundant cover for a given function.
  - Prime cover leads to min number of inputs to each product term.
  - Min irredundant cover leads to min number of product terms.
- Quine-McCluskey (QM) method (1960's) finds a minimum prime and irredundant cover.
  - Step 1: List all minterms of on-set: O(2^n) n = #inputs
  - Step 2: Find all primes
  - Step 3: Construct minterms vs primes table
  - Step 4: Find a min set of primes that covers all the minterms: O(2^m) m = #primes

QM Example (Step 1)

- F = a' b' c' + a b' c' + a b' c + a b c + a' b c
- List all on-set minterms

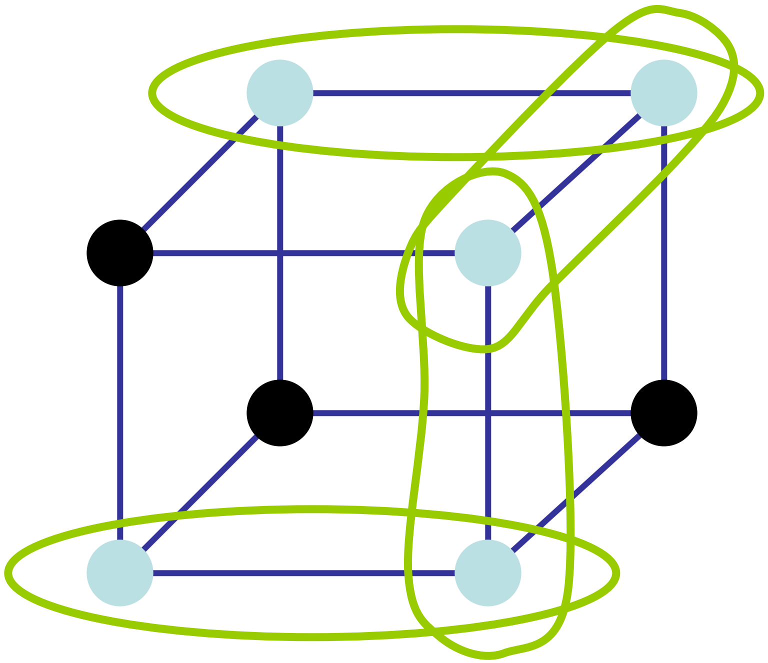
Minterms
a' b' c'
a b' c'
a b' c
a b c
a' b c



QM Example (Step 2)

- F = a’ b’ c’ + a b’ c’ + a b’ c + a b c + a’ b c
- Find all primes.

primes	b’ c’	a b’	a c	b c



QM Example (Step 3)

- F = a’ b’ c’ + a b’ c’ + a b’ c + a b c + a’ b c
- Construct minterms vs primes table (prime implicant table) by determining which cube is contained in which prime. X at row i, column j means that cube in row i is contained by prime in column j.

	b’ c’	a b’	a c	b c
a’ b’ c’	X			
a b’ c’	X	X		
a b’ c		X	X	
a b c			X	X
a’ b c				X

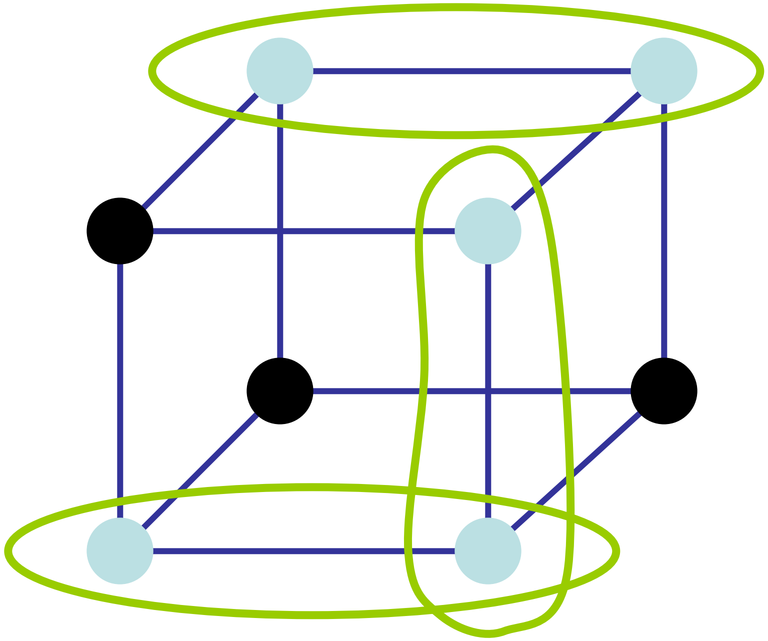
QM Example (Step 4)

- F = a' b' c' + a b' c' + a b' c + a b c + a' b c
- Find a minimum set of primes that covers all the minterms

  "Minimum column covering problem"

	b' c'	a b'	a c	b c
a' b' c'	X			
a b' c'	X	X		
a b' c		X	X	
a b c			X	X
a' b c				X

Essential primes



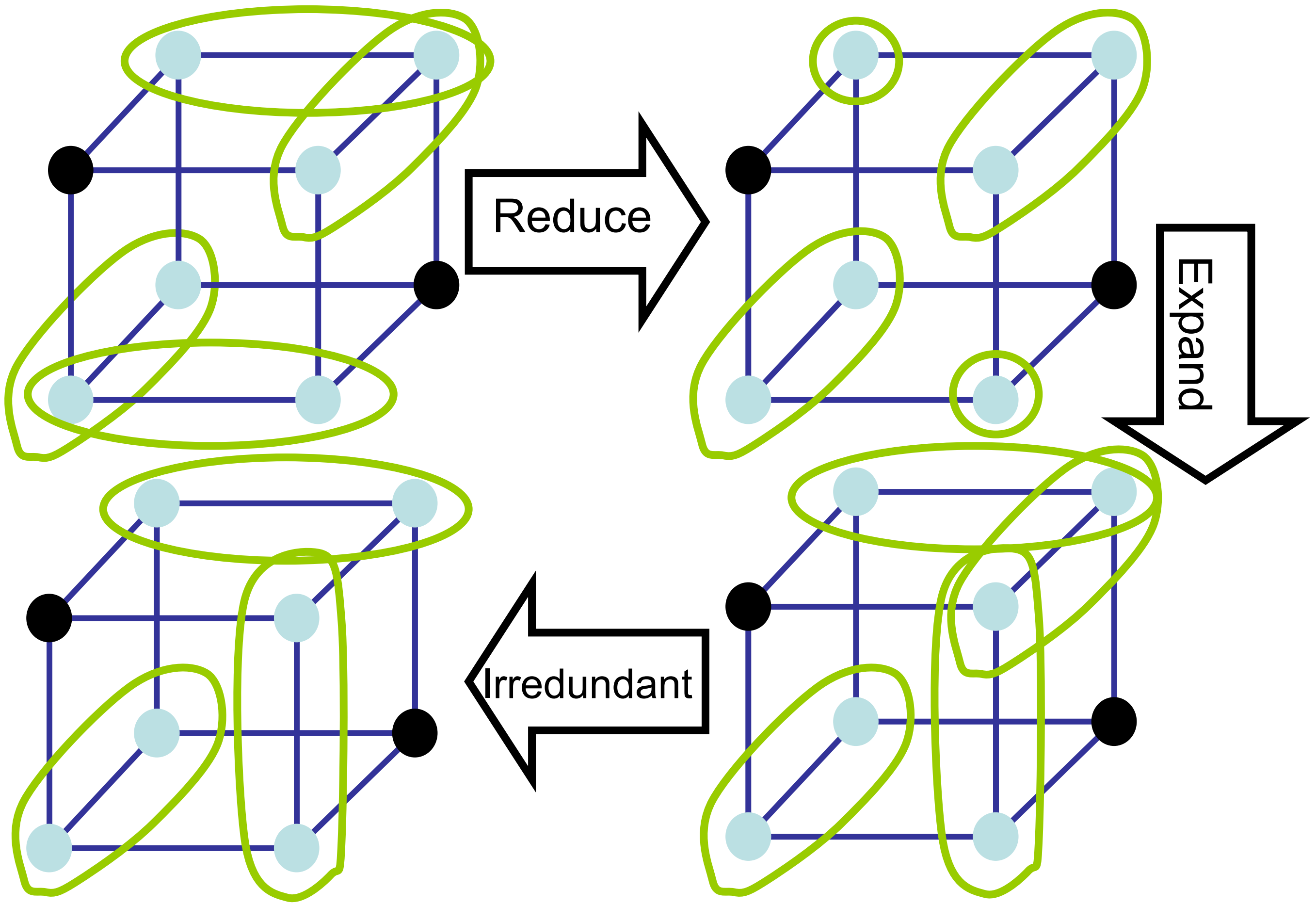
How to find prime implicants ?

- For 3 input functions, you can visualize them in a cube. What about more than 3 inputs ?
- E.g., **f = m(0, 3, 7, 12, 13, 14, 15) (4 inputs)**

Heuristic Logic Minimization

- Provide irredundant covers with small sizes
  - Much faster than exact minimization (e.g., Quine-McCluskey)
- Basic approach
  - Start from initial cover
  - Modify cover under consideration →Size of the cover decreases with each iteration
- Basic routines (Espresso)
  - Expand: Make cubes prime; Remove covered cubes
  - Reduce: Reduce size of each cube while preserving cover
  - Irredundant: Make cover irredundant

ESPRESSO ILLUSTRATED



ESPRESSO(F) {
do {
reduce(F);
expand(F);
irredundant(F);
} while (fewer terms in F);
}

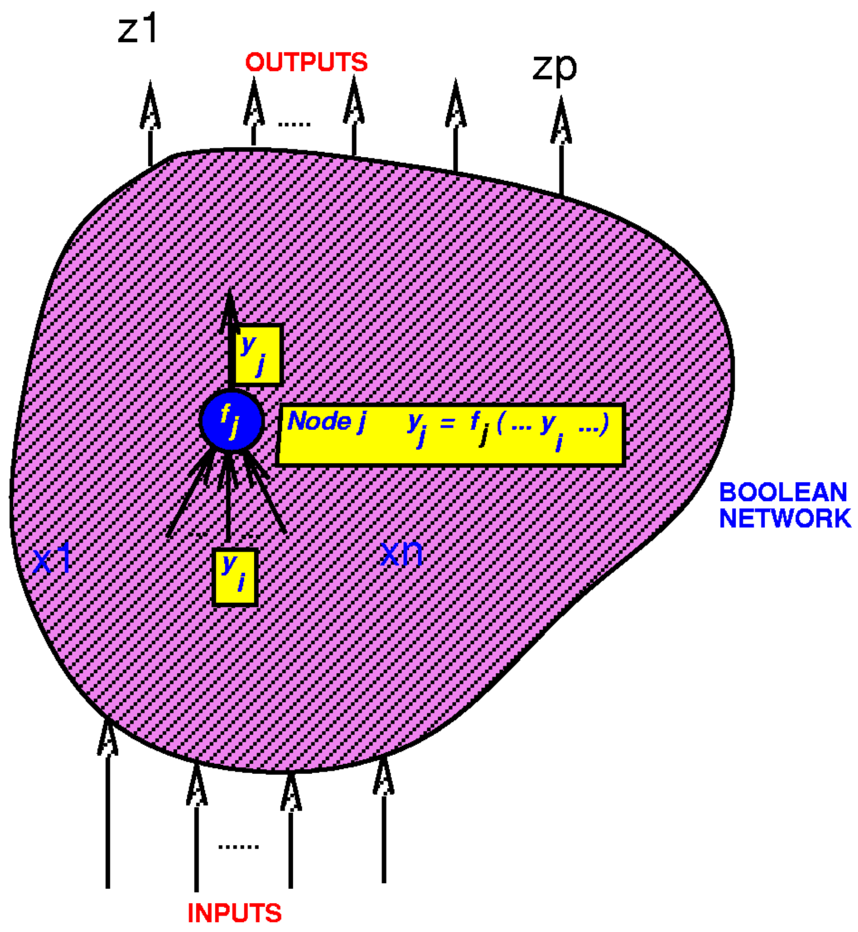
Representation: Boolean Network

Boolean network:

- directed acyclic graph (DAG)
- node logic function representation $f_j(x,y)$
- node variable y_j: $y_j = f_j(x,y)$
- edge (i,j) if f_j depends explicitly on y_i

Inputs $x = (x_1, x_2, \ldots, x_n)$

Outputs $z = (z_1, z_2, \ldots, z_p)$



Node Representation: Sum of Products (SOP)

- Example: abc'+a'bd+b'd'+b'e'f (sum of cubes)
- Advantages:
  - easy to manipulate and minimize
  - many algorithms available (e.g. QM)
  - two-level theory applies
- Disadvantages:
  - Not representative of logic complexity. For example

    f=ad+ae+bd+be+cd+ce f'=a'b'c'+d'e'

    - These differ in their implementation by an inverter.
  - hence not easy to estimate logic; difficult to estimate progress during logic manipulation

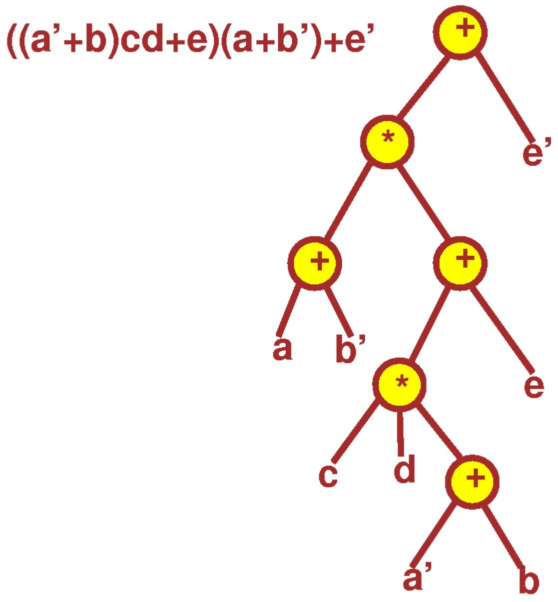
Factored Forms

- Example: $(ad+b'c)(c+d'(e+ac'))+(d+e)fg$
- Advantages
  - good representative of logic complexity
    $f=ad+ae+bd+be+cd+ce \qquad f'=a'b'c'+d'e' \rightarrow$
    $f=(a+b+c)(d+e)$
  - in many designs (e.g. complex gate CMOS) the implementation of a function corresponds directly to its factored form (standard cells)
  - good estimator of logic implementation complexity
- Disadvantages
  - not as many algorithms available for manipulation
  - hence usually just convert into SOP before manipulation

Factored Forms

Factored forms can be graphically represented as labeled trees, called factoring trees, in which each internal node including the root is labeled with either + or $\times$, and each leaf has a label of either a variable or its complement.

Example: factoring tree of *((a'+b)cd+e)(a+b')+e'*

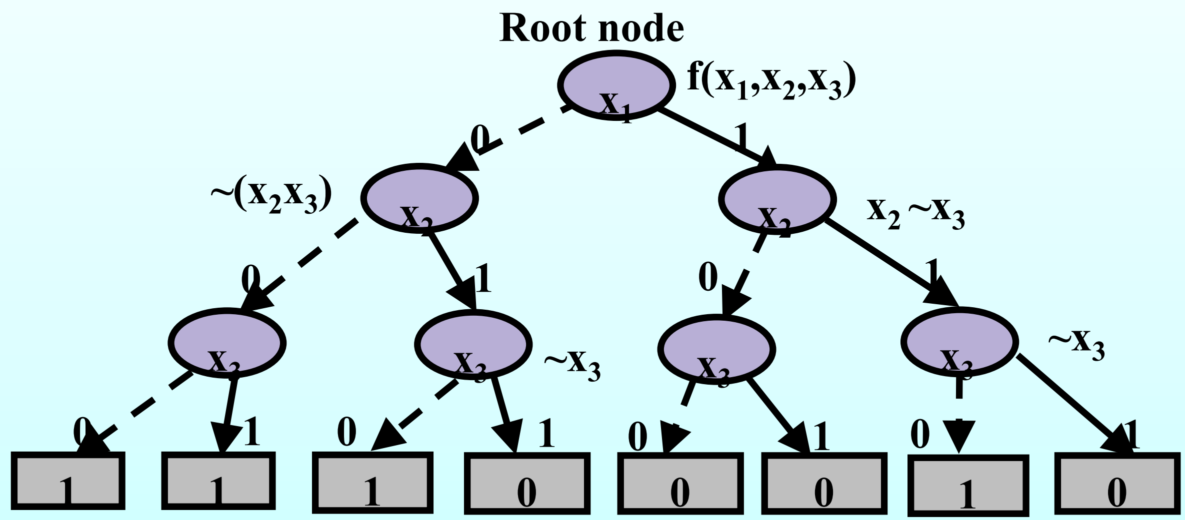


Binary Decision Diagrams

- A Binary Decision Diagram (BDD) is a directed acyclic graph
  1. Each vertex represents a decision on a variable
  2. The value of the function is found at the leaves
  3. Each path from root to leaf corresponds to a row in the truth table

– Many logic functions can be represented compactly - usually better than SOP's

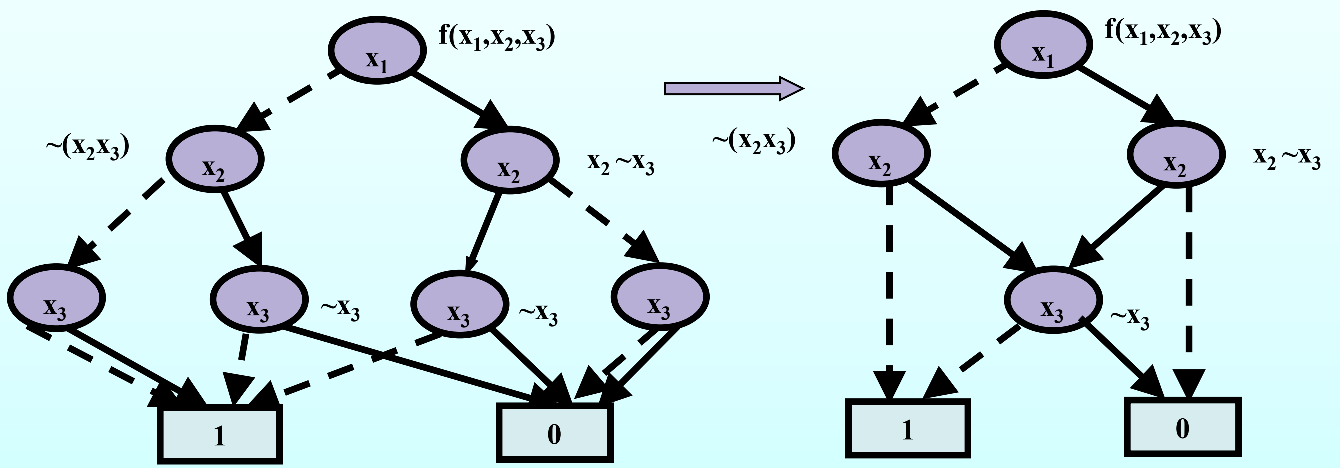
x_1 x_2 x_3	$f(x_1x_2x_3)$
0 0 0	1
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	0
1 0 1	0
1 1 0	1
1 1 1	0



$$f(x_1, x_2, x_3) = \sim x_1 \sim x_2 \sim x_3 + \sim x_1 \sim x_2 + \sim x_1 x_2 \sim x_3 + x_1 x_2 \sim x_3$$

ROBDDs – Reduced Ordered BDDs

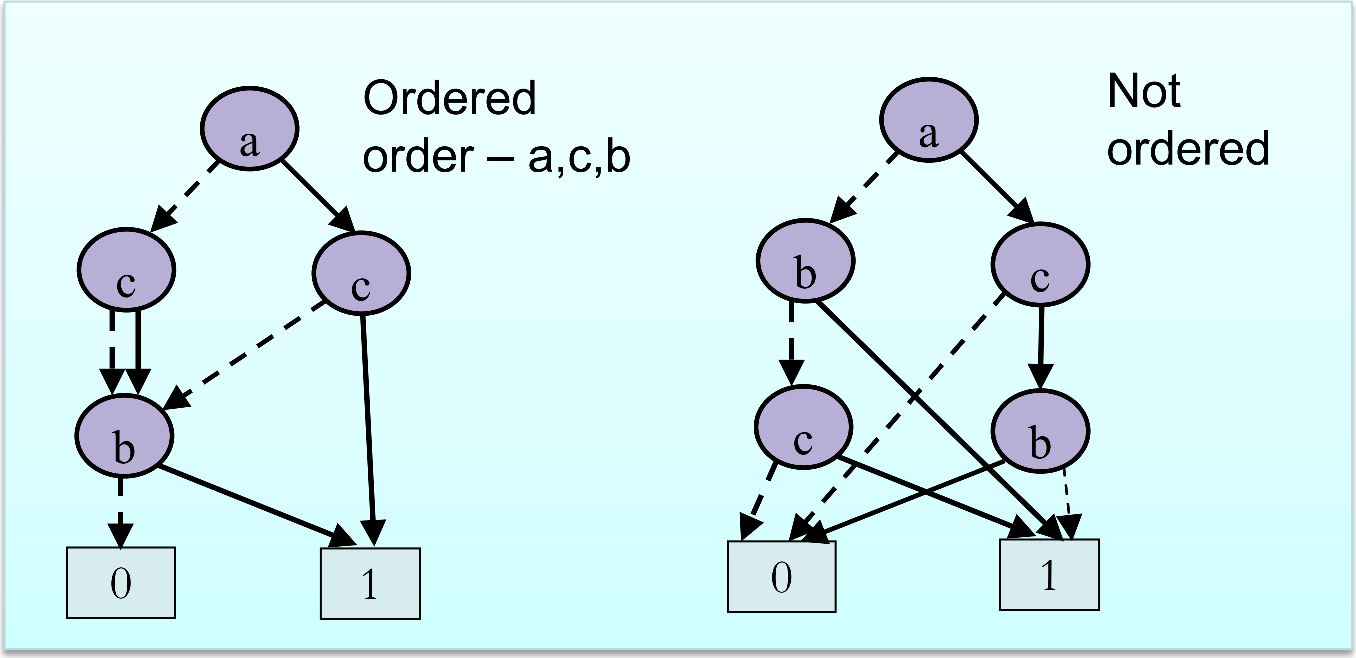
- ROBDD:
  - Directed acyclic graph (DAG)
  - one root node, two terminals 0, 1
  - each node, two children, and a variable
  - Reduced:
    - any node with two identical children is removed
    - two nodes with isomorphic BDD's are merged
  - Ordered:
    - Splitting variables always follow the same order along all paths



$f(x_1, x_2, x_3) = \sim x_1 \sim x_2 + \sim x_1 x_2 \sim x_3 + x_1 x_2 \sim x_3$

Ordered vs. Not

- Size of BDD critically dependent on variable ordering
  - for a good ordering, BDDs remain reasonably small for complicated functions
- ROBDD is canonical given a variable ordering → a good replacement for truth tables



5 min break

Technology-Independent Optimization: Bag of Tricks

- Two-level minimization (also called simplify)
  - Use heuristic minimizer like Espresso

$u = q'c + qc' + qc \rightarrow u = q + c;$

- Constant propagation (also called sweep)
  - Boolean minimization may lead to dissolution of certain section of code into constants

    f = a b + c; b = 1 => f = a + c

- Collapsing (also called elimination)
  - Eliminate one function from the network

$f = ga+g'b \qquad g = c+d$

$\Downarrow$

$f = ac+ad+bc'd'$

- Factoring (series-parallel decomposition)

$f = ac+ad+bc+bd+e \Rightarrow f = (a+b)(c+d)+e$

More Technology-Independent Optimization

- Decomposition (single function)
  - Break a function into smaller ones

    $f = abc+abd+a'c'd'+b'c'd' \Rightarrow f = xy + x'y'; \quad x = ab \; ; \; y = c+d$
- Extraction (multiple functions)
  - Find a common sub-expression of two (or more) expressions

    $f = (az+bz')cd+e \quad g = (az+bz')e' \quad h = cde$

    $\Downarrow$

    $f = xy+e \quad g = xe' \quad h = ye \quad x = az+bz' \quad y = cd$
- Substitution
  - Simplify a local function by using an additional input (that already exists in the network elsewhere)

    $g = a+b \qquad f = a+bc$

    $\Downarrow$

    $f = g(a+c)$

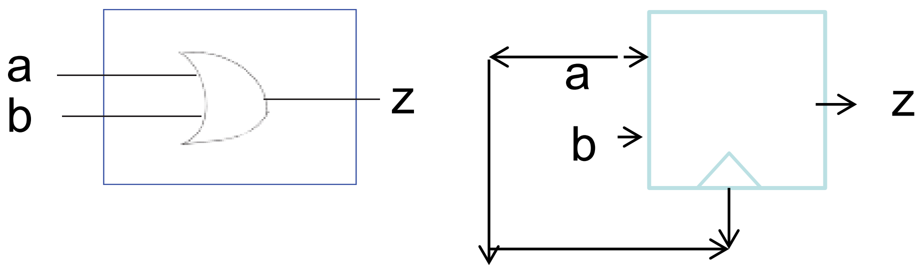
Boolean vs. Algebraic Manipulation

- Boolean methods for multilevel synthesis
  - Exploit properties of Boolean functions (e.g., a a' = 0)
  - Use *don't care* conditions
  - Computationally intensive
- Algebraic methods
  - Use polynomial abstraction of logic function
  - Simpler, faster, weaker
  - Widely used
- Example: Boolean vs. Algebaric factoring

  Consider f = a b' + a c' + b a' + b c' + c a' + c b'

  - Algebraic: f = a (b' + c') + a' (b + c) + b c' + c b'
  - Boolean: f = (a + b + c) (a' + b' + c')

Logical Equivalence Checking Using BDDs



– Logical equivalence checking (i.e., checking if two netlists are implementing the same function)
  - Very useful to compare RTL to gate-level netlist or compare gate-level netlists pre and post layout
– It is just graph isomorphism check on the corresponding canonical ROBDDs
  - Graph isomorphism: Two graphs which contain the same number of graph vertices connected in the same way are said to be isomorphic
  - E.g., Left circuit above is a OR b. Right one is MUX(a, b, a). Draw the ROBDD for MUX and OR and check if they are "same".

Technology Mapping

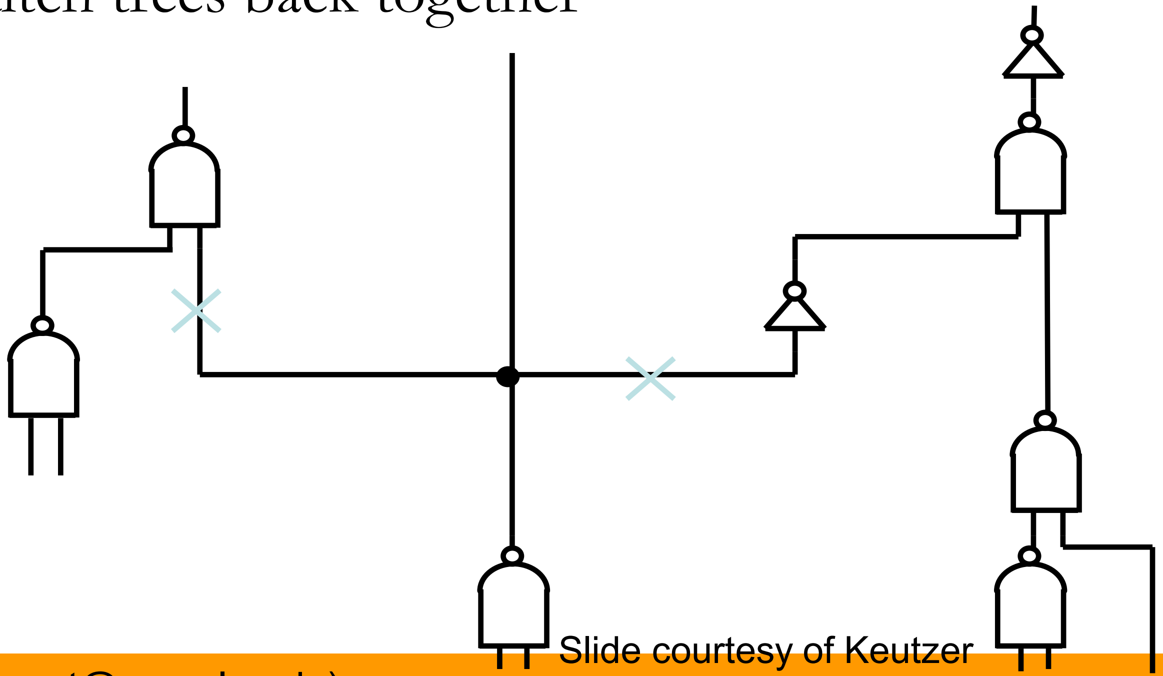
Input
- Technology independent, optimized logic network
- Description of the gates in the library with their cost

Output
- Netlist of gates (from library) which minimizes total cost

General Approach
- Construct a subject DAG for the network
- Represent each gate in the target library by pattern DAG's
- Find an optimal-cost covering of subject DAG using the collection of pattern DAG's
- Canonical form: 2-input NAND gates and inverters

DAG Covering

- DAG covering is an NP-hard problem
- Solve the sub-problem optimally
  - Partition DAG into a forest of trees
    - Break at every node with fanout > 1
  - Cover each tree optimally using dynamic programming
  - Stitch trees back together



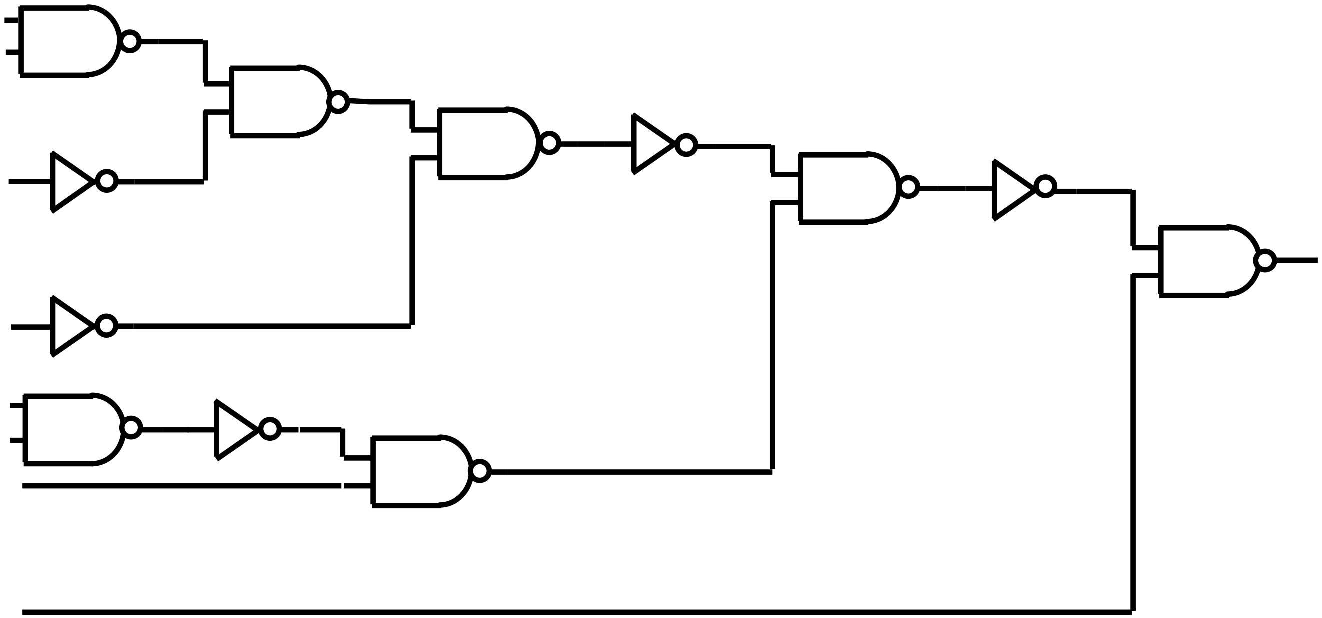
Slide courtesy of Keutzer

Tree Covering Algorithm

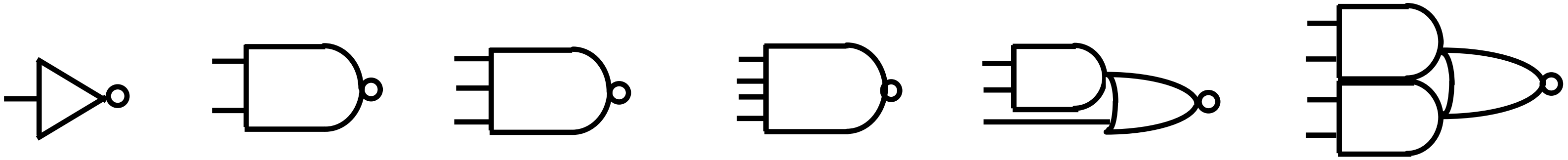
- Transform netlist and libraries into canonical forms
  – 2-input NANDs and inverters
- Visit each node in BFS from inputs to outputs
  – Find all candidate matches at each node N
    - Match is found by comparing topology only (no need to compare functions)
  – Find the optimal match at N by computing the new cost
    - New cost = cost of match at node N + sum of costs for matches at children of N
  – Store the optimal match at node N with cost
- Optimal solution is guaranteed if cost is area
- Complexity = O(n) where n is the number of nodes in netlist

Tree Covering Example

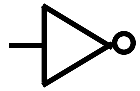
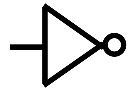
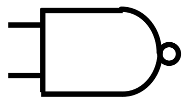
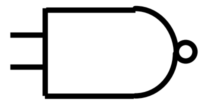
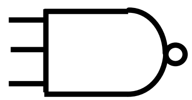
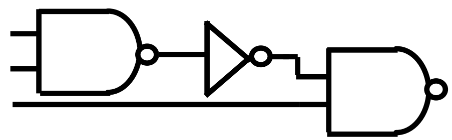
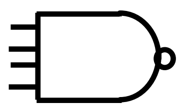
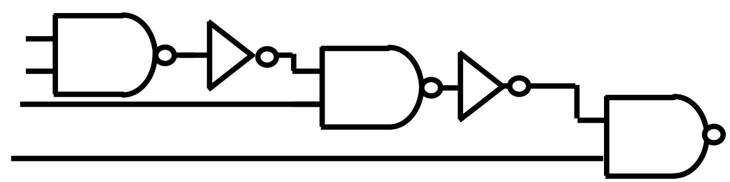
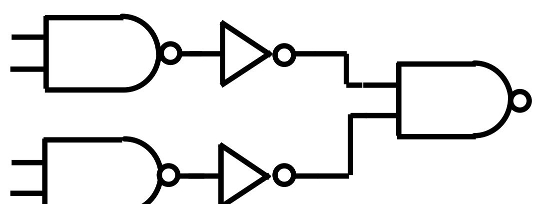
Find an ``optimal'' (in area, delay, power) mapping of this circuit



into the technology library (simple example below)

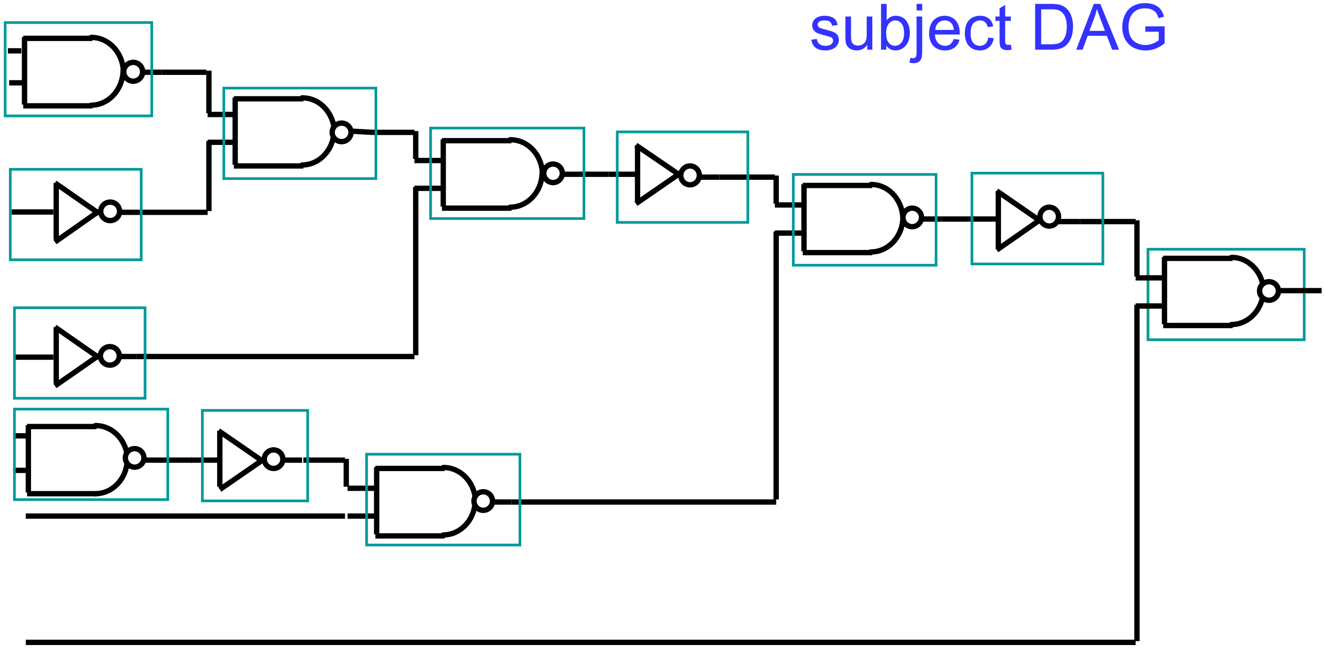


Elements of a library - 1

	Element/Area Cost		Tree Representation (normal form)
INVERTER	2		
NAND2	3		
NAND3	4		
NAND4	5		 

Trivial Covering

subject DAG

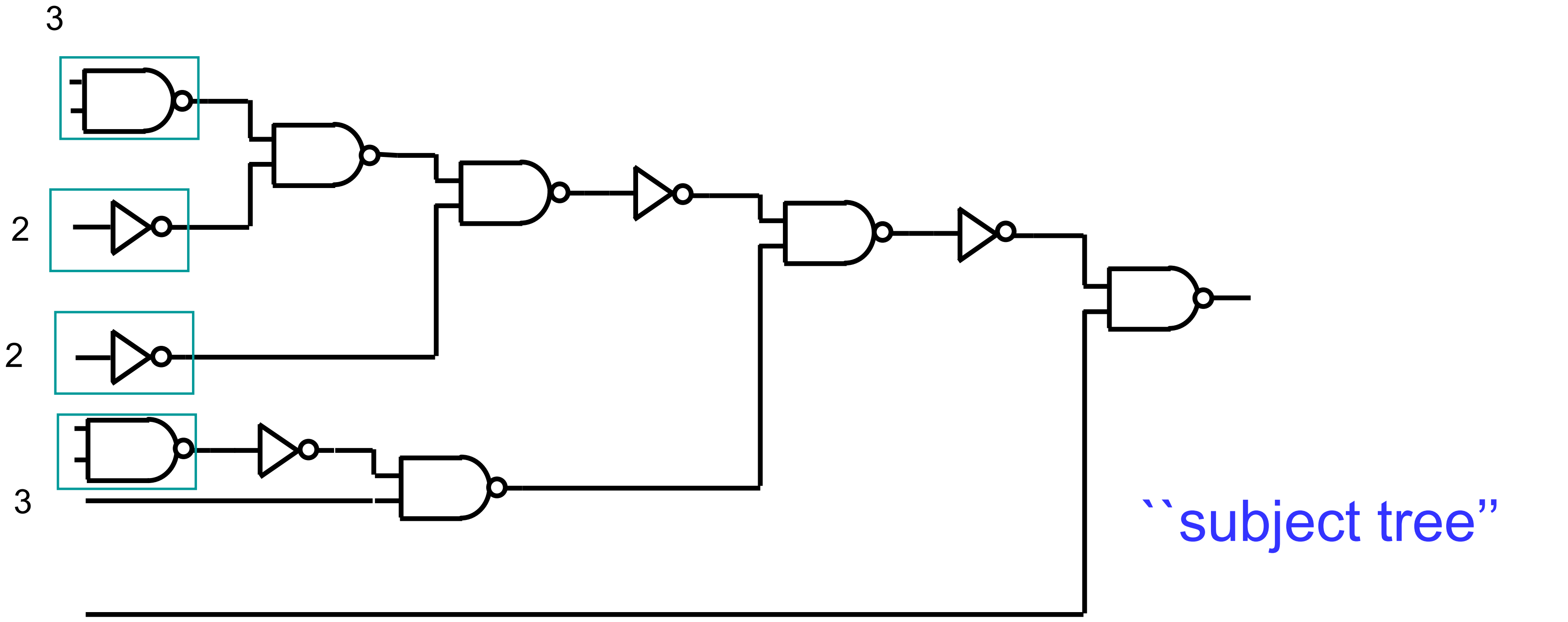


7	NAND2 (3) =	21
5	INV (2) =	10

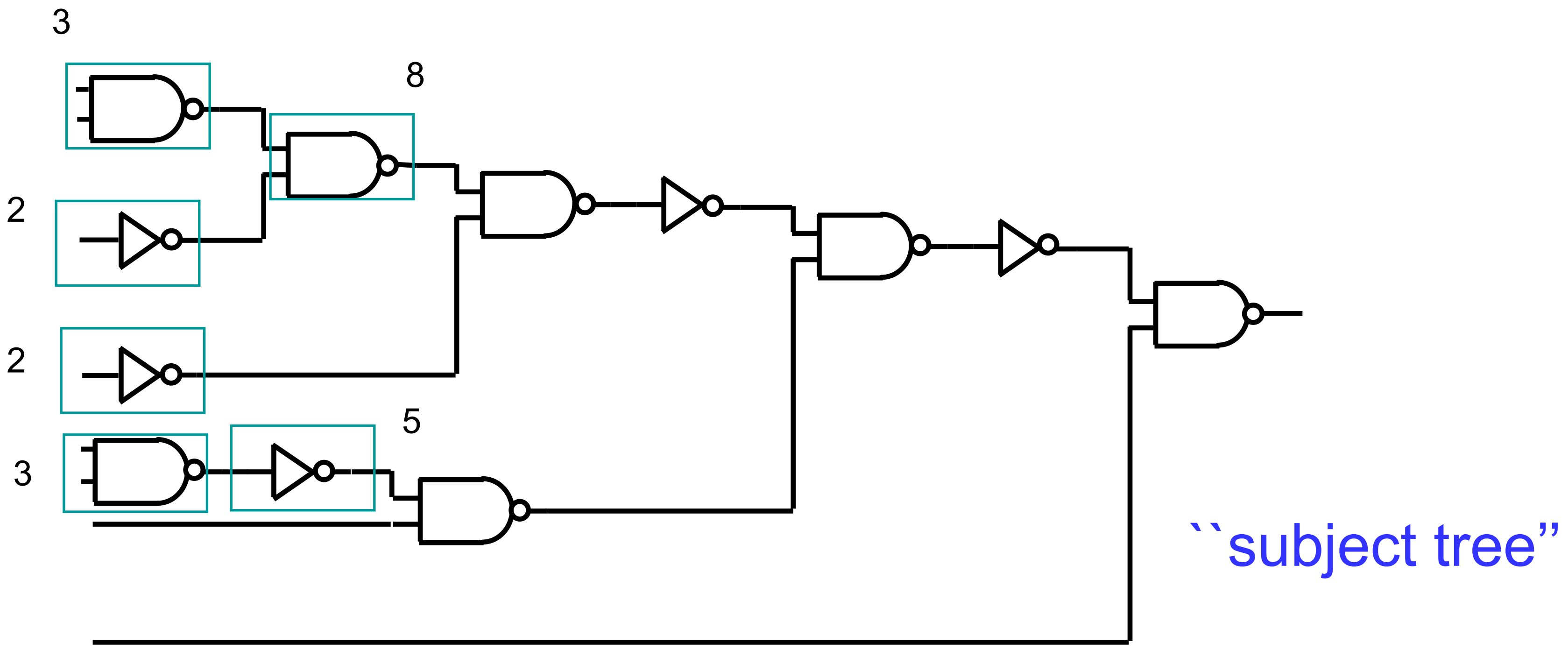
Area cost 31

Can we do better with tree covering?

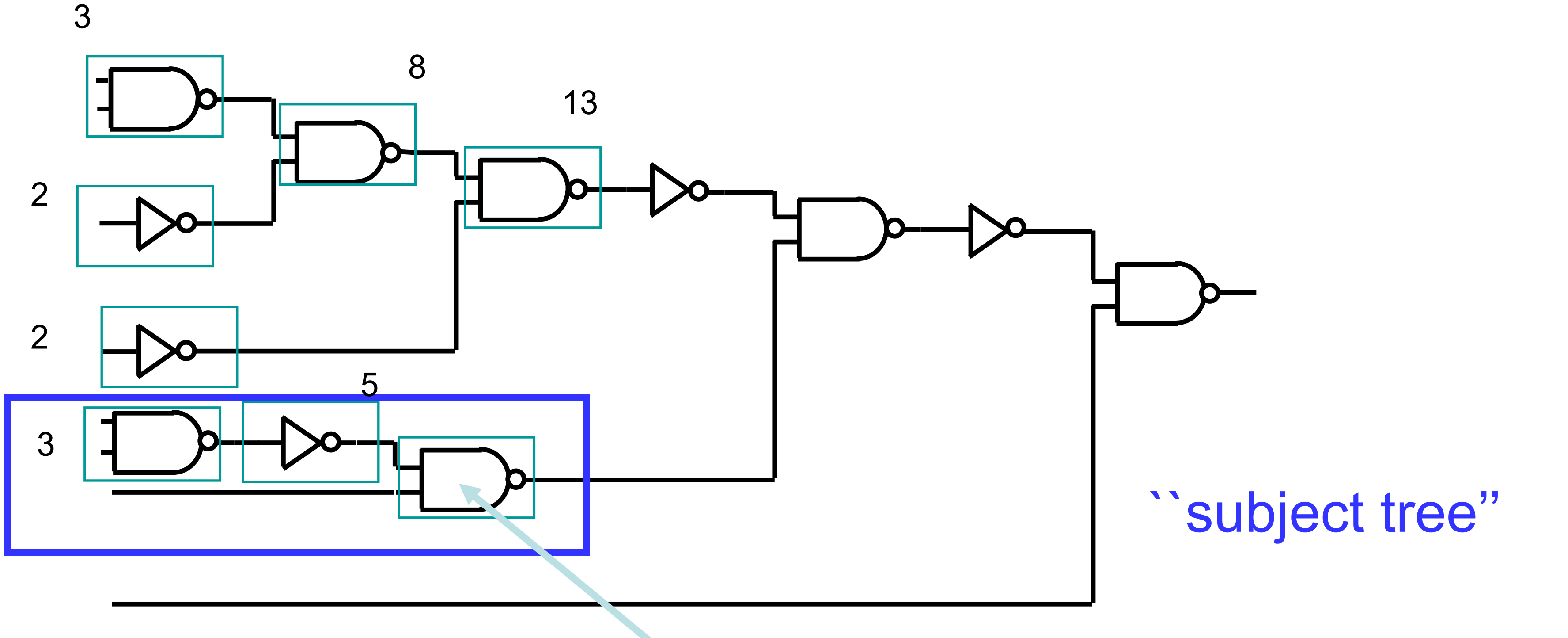
Optimal tree covering - 1



Optimal tree covering - 2



Optimal tree covering - 3



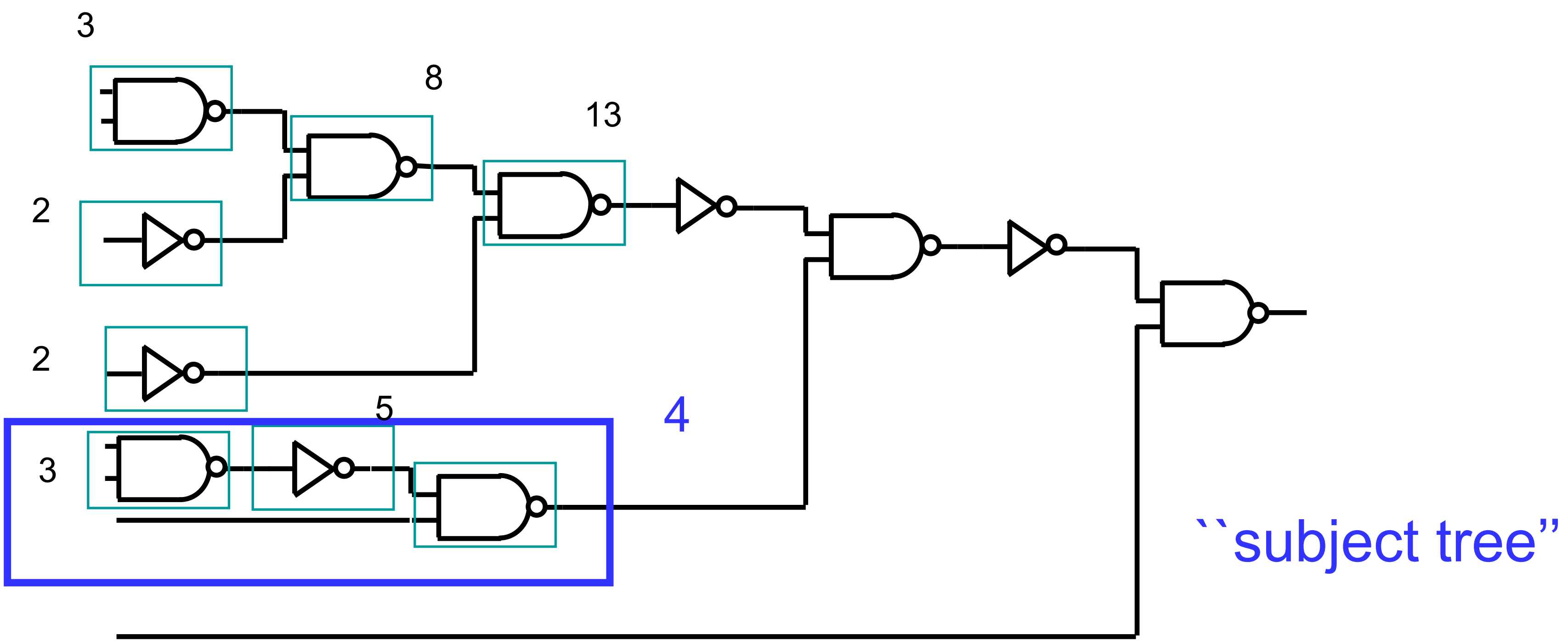
Cover with ND2 or ND3 ?

1 NAND2	3
+ subtree	5

1 NAND3 = 4

Area cost 8

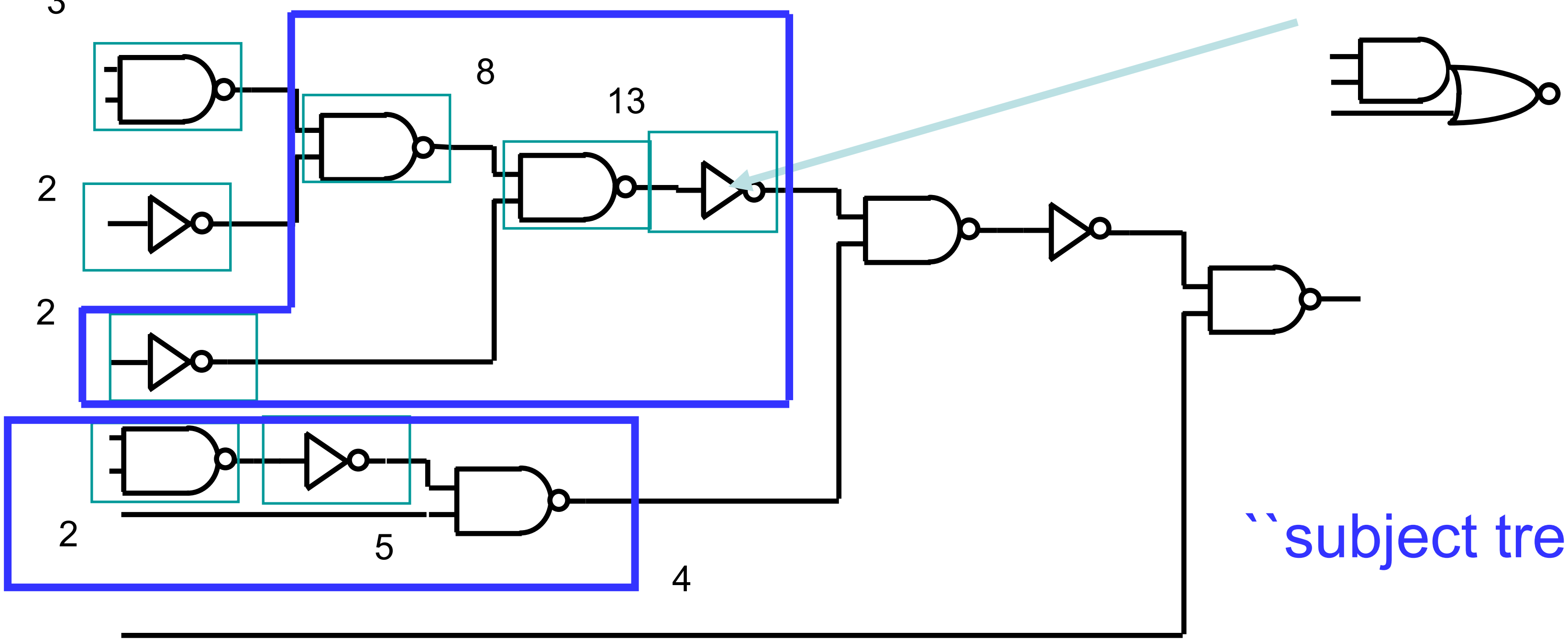
Optimal tree covering – 3b



Label the root of the sub-tree with optimal match and cost

Optimal tree covering - 4

Cover with INV or AO21 ?



``subject tree''

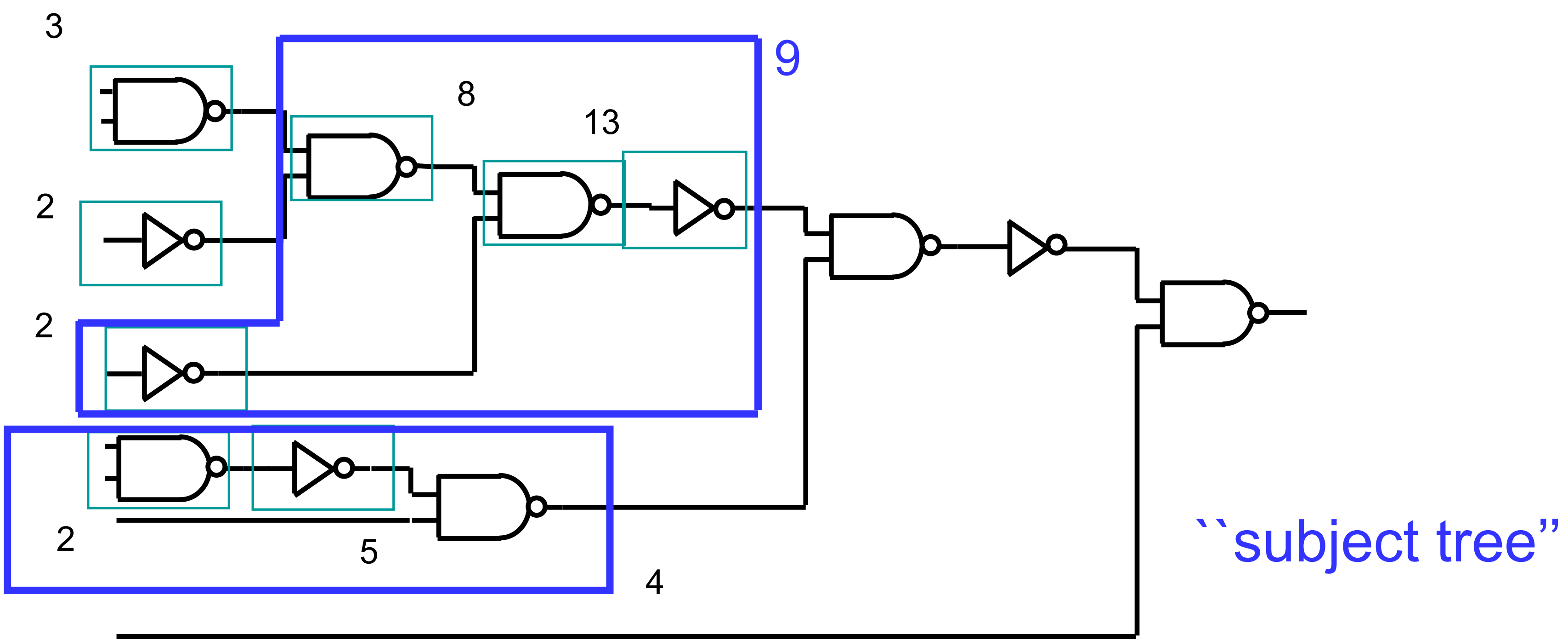
1 Inverter 2
+ subtree 13

Area cost 15

1 AO21 4
+ subtree 1 3
+ subtree 2 2

Area cost 9

Optimal tree covering – 4b

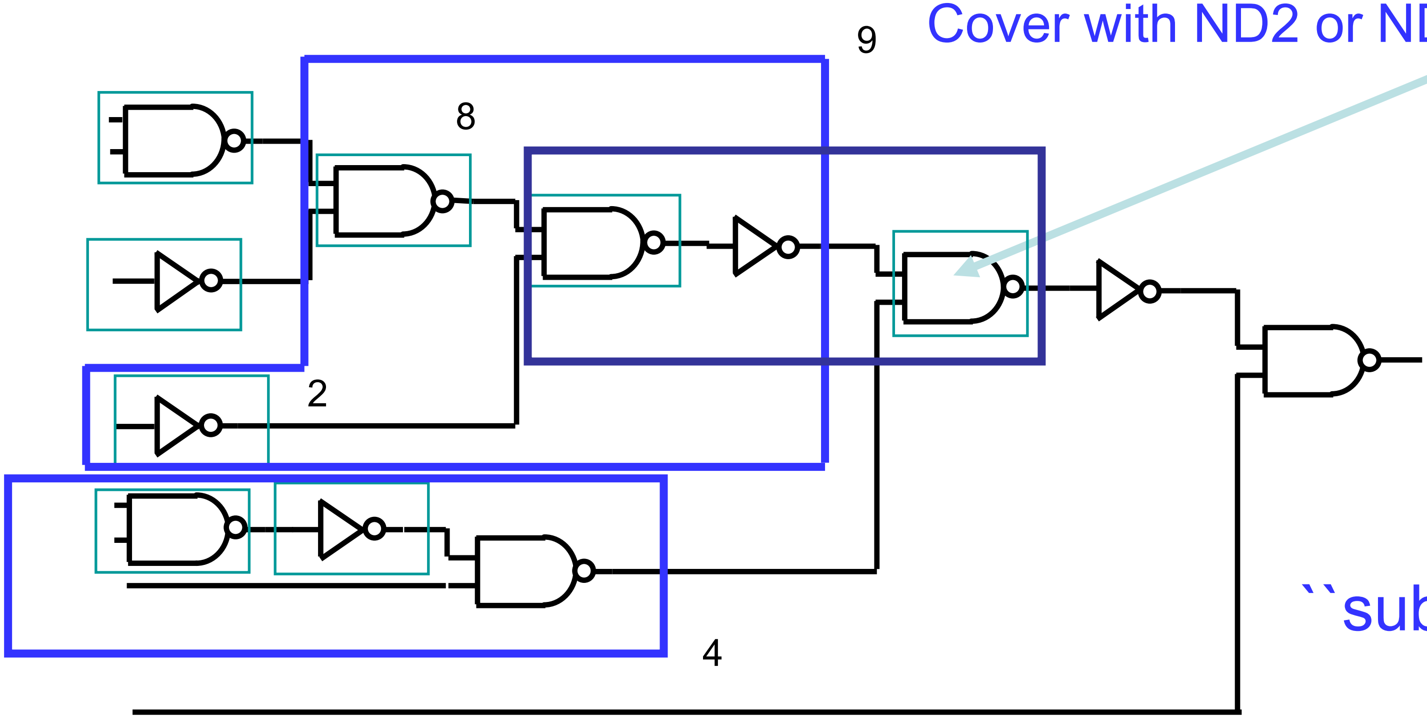


Label the root of the sub-tree with optimal match and cost

Zoom poll: What is the next set of choices ?

Optimal tree covering - 5

Cover with ND2 or ND3 ?



``subject tree''

NAND2

subtree 1	9
subtree 2	4
1 NAND2	3

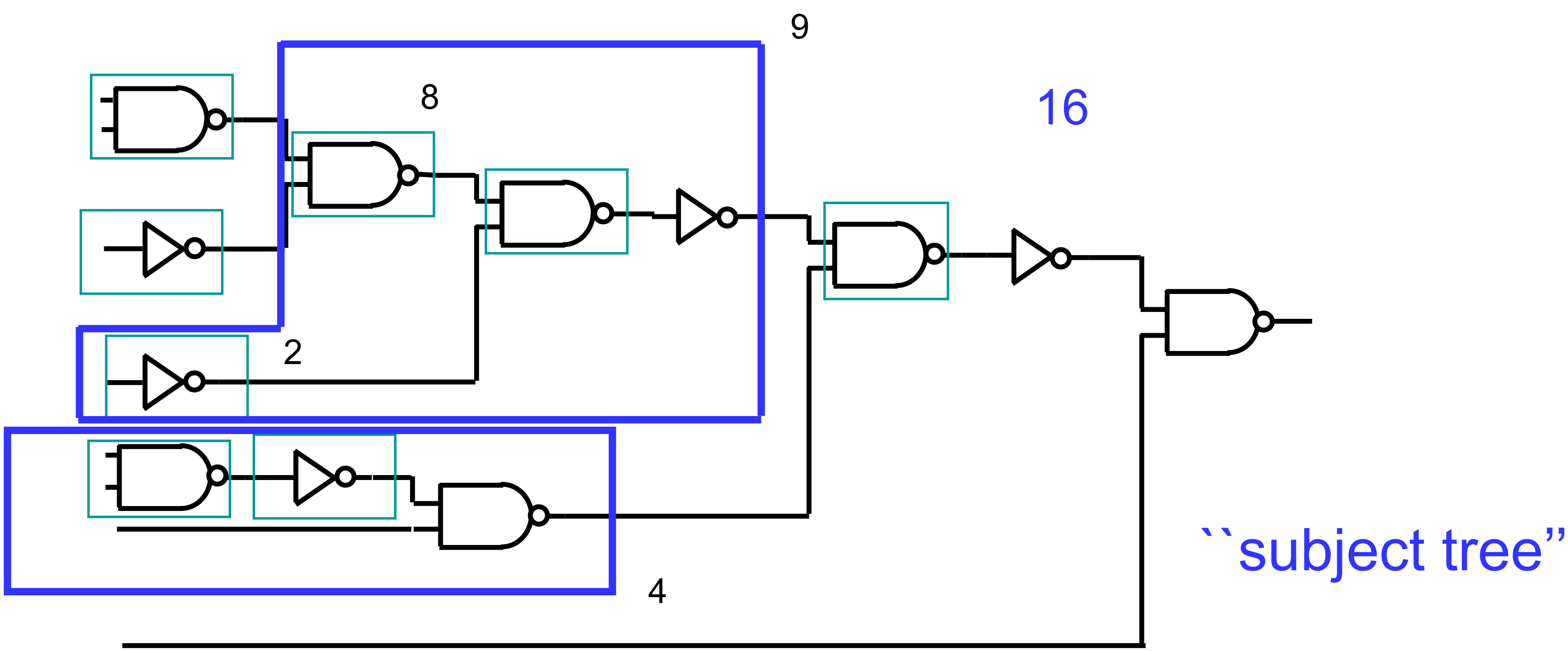
Area cost 16

NAND3

subtree 1	8
subtree 2	2
subtree 3	4
1 NAND3	4

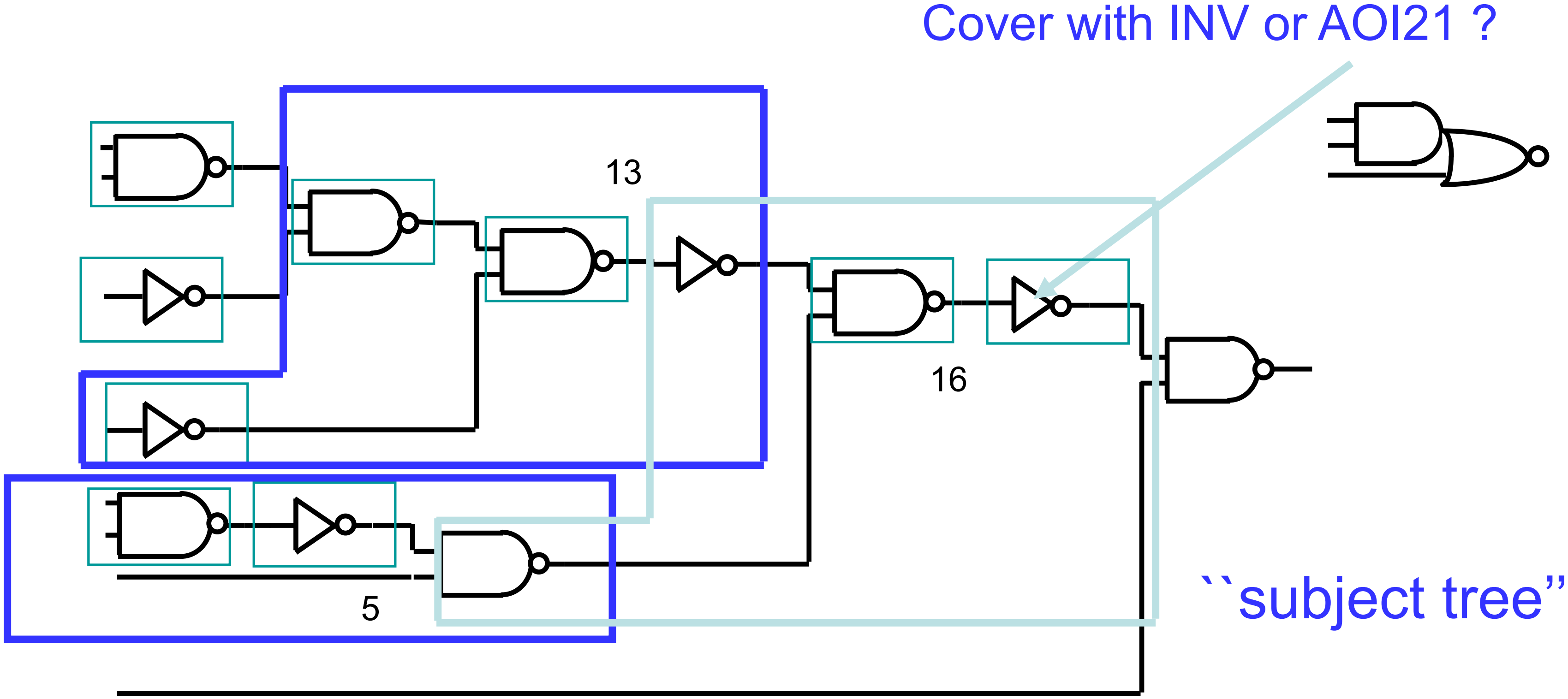
Area cost 18

Optimal tree covering – 5b



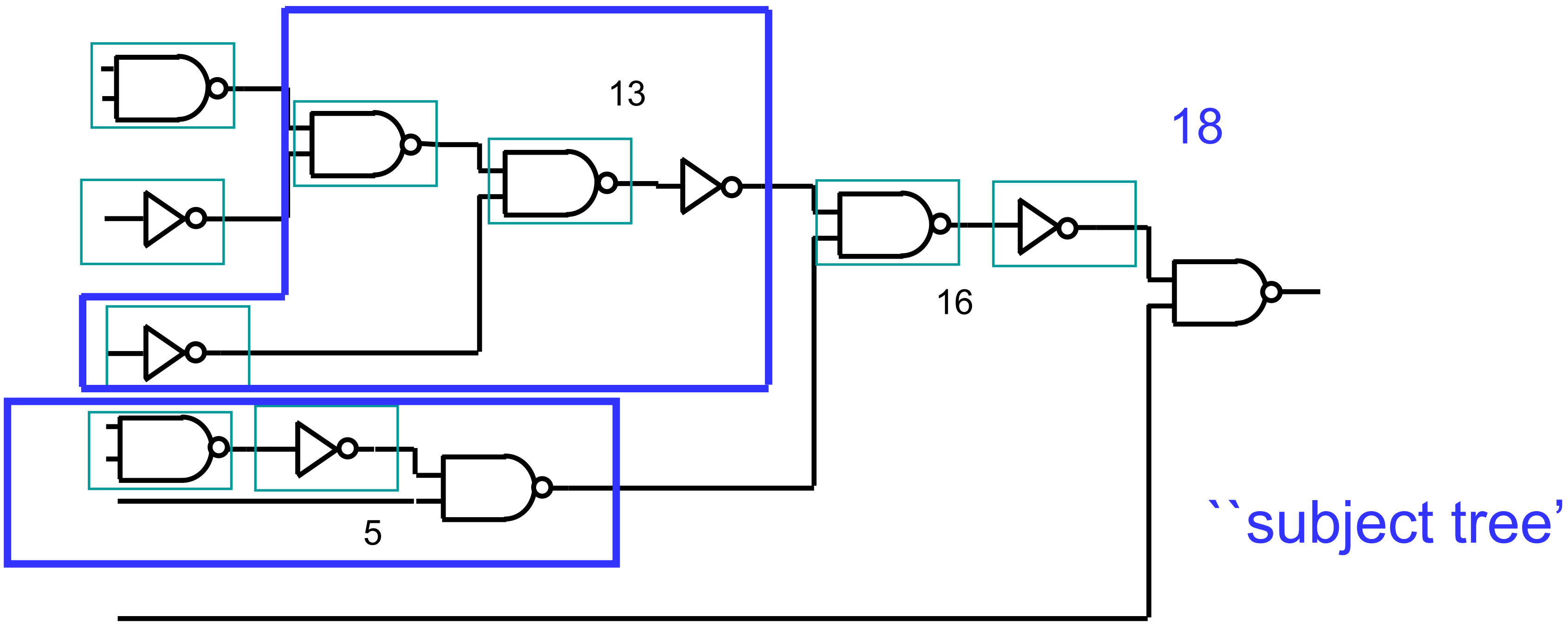
Label the root of the sub-tree with optimal match and cost

Optimal tree covering - 6



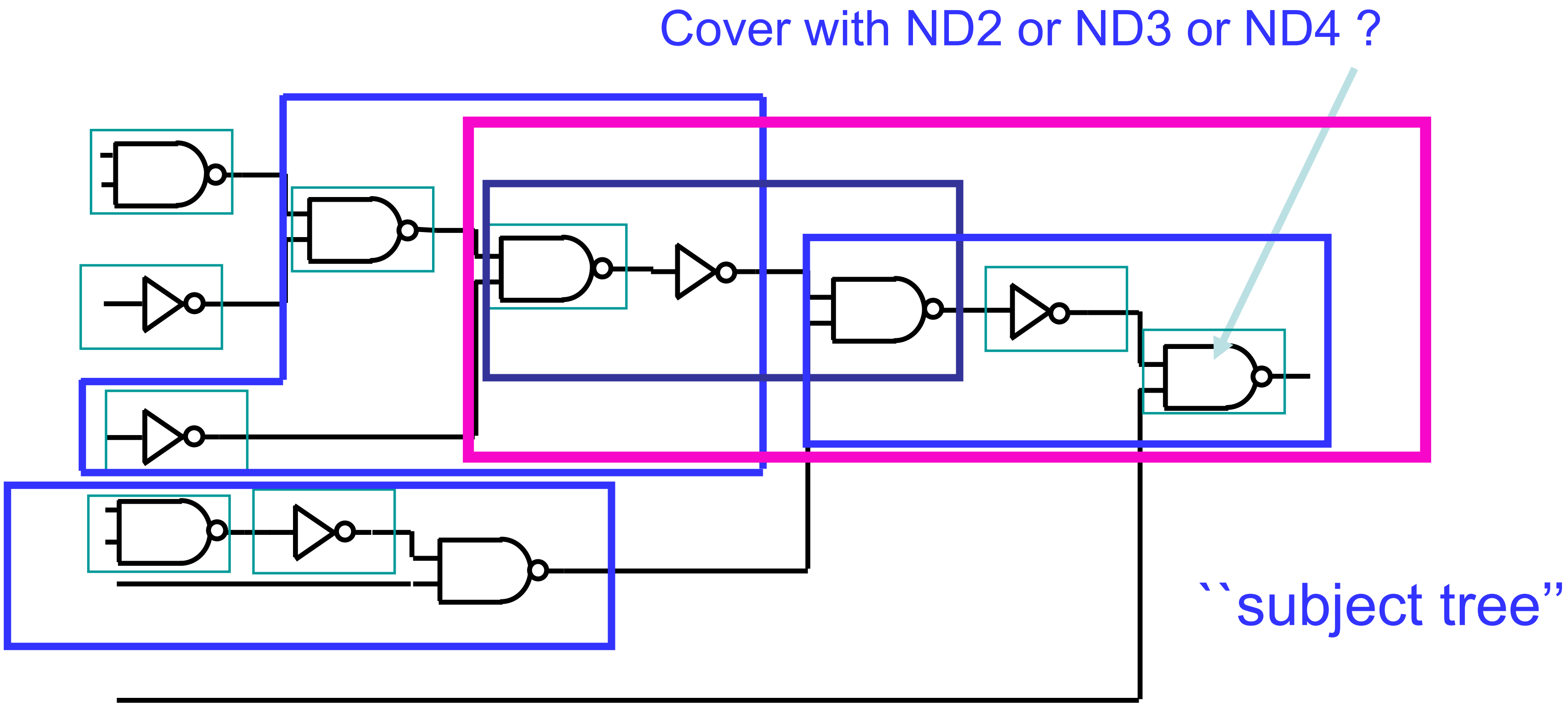
INV	subtree 1 1 INV	16 2	AOI21	subtree 1 subtree 2 1 AOI21	13 5 4
	Area cost 18			Area cost 22	

Optimal tree covering – 6b

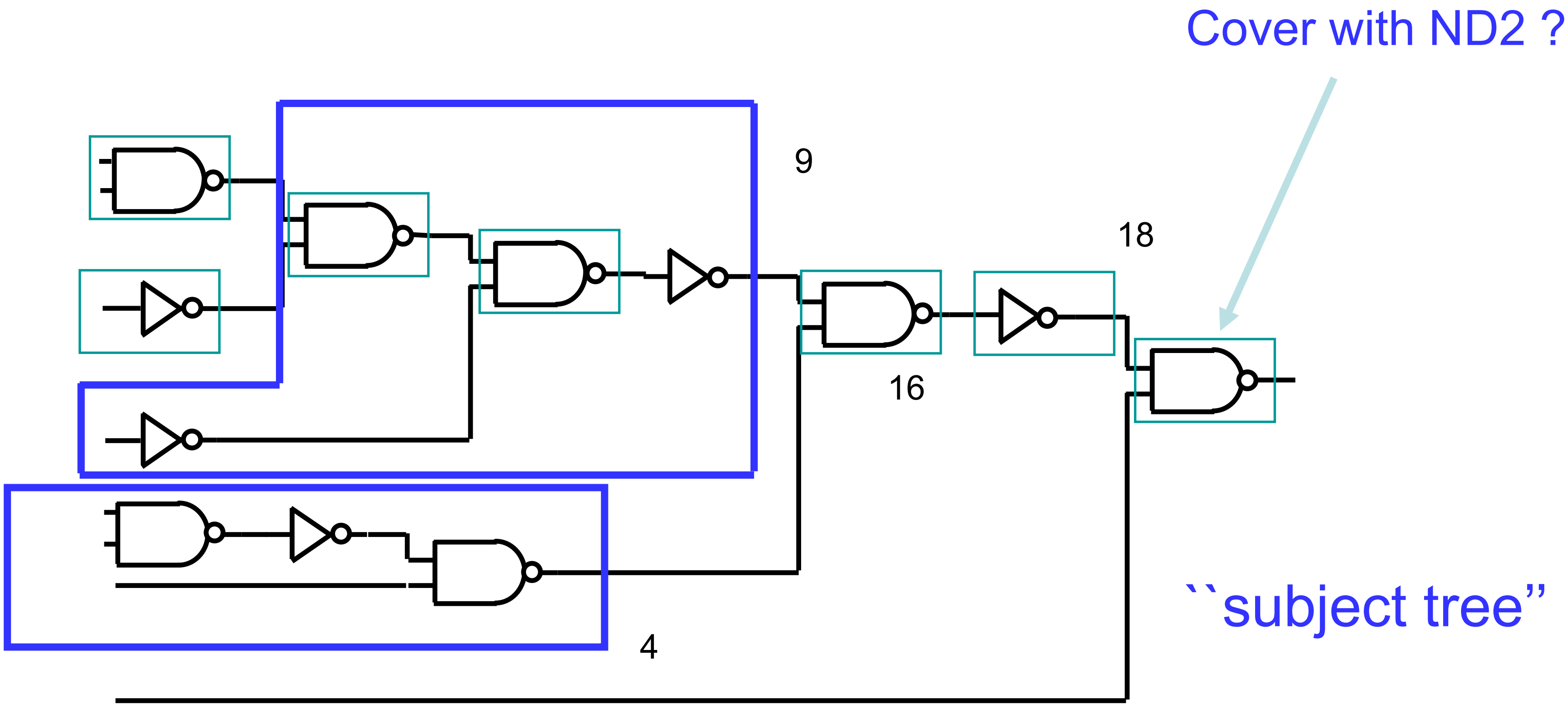


Label the root of the sub-tree with optimal match and cost

Optimal tree covering - 7



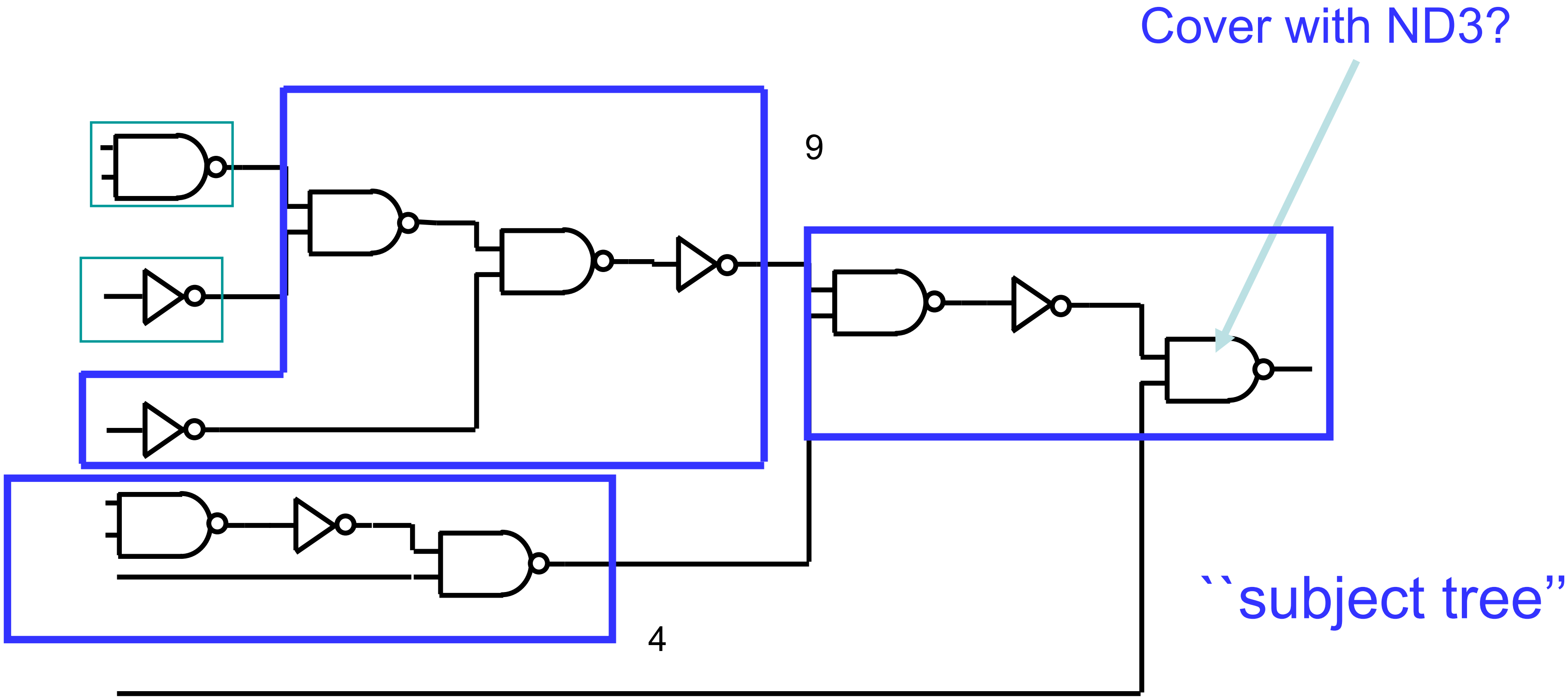
Cover 1 - NAND2



subtree 1 18
subtree 2 0
1 NAND2 3

Area cost 21

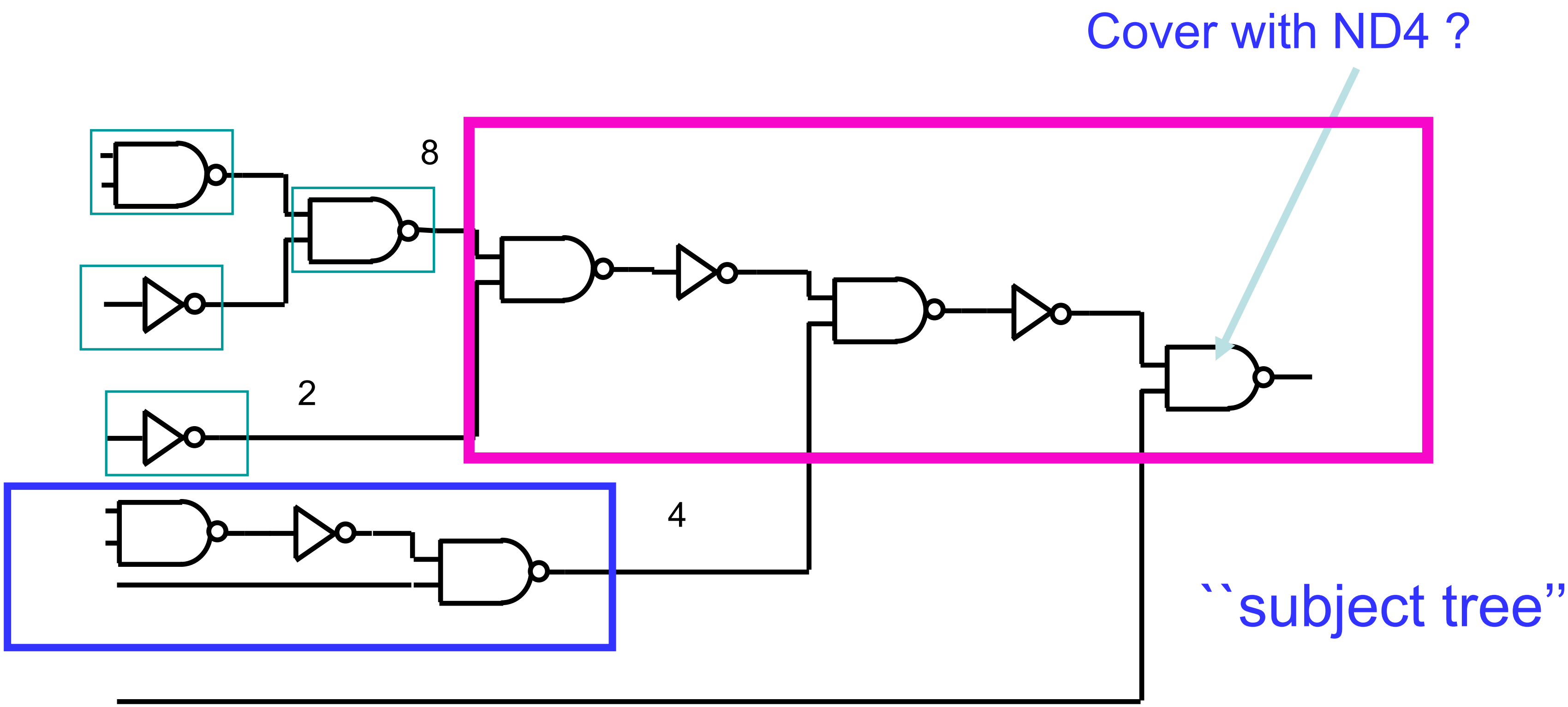
Cover 2 - NAND3



subtree 1 9
subtree 2 4
subtree 3 0
1 NAND3 4

Area cost 17

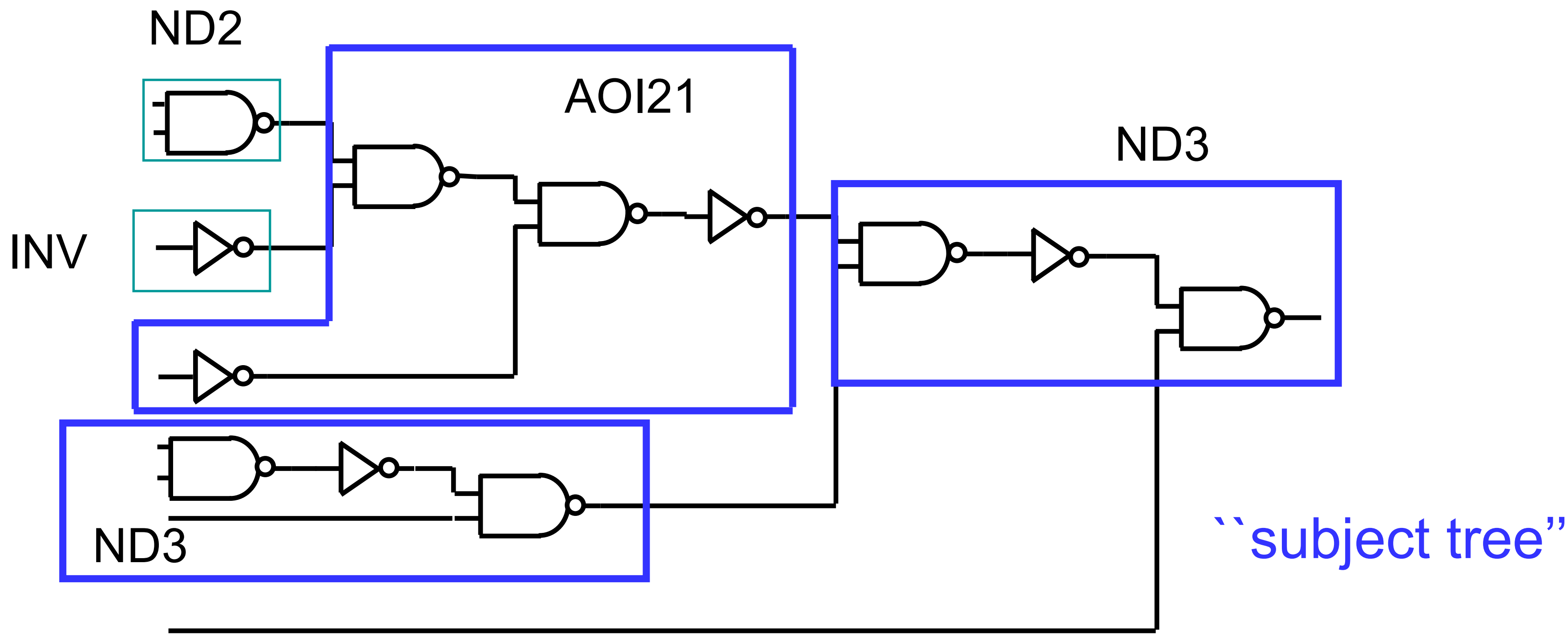
Cover - 3



subtree 1 8
subtree 2 2
subtree 3 4
subtree 4 0
1 NAND4 5

Area cost 19

Optimal Cover was Cover 2

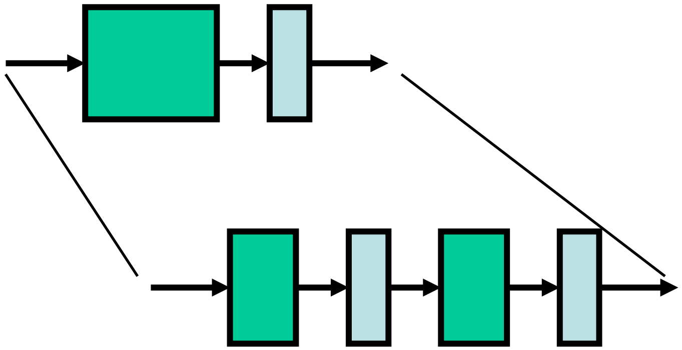


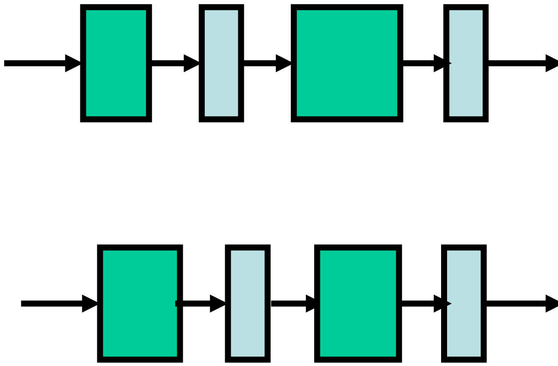
INV	2
ND2	3
2 ND3	8
AOI21	4

Area cost 17

Example Sequential Optimizations

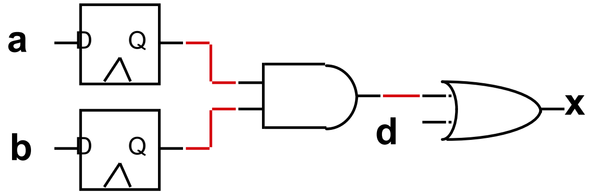
- Pipelining
  - split combinational logic into multiple cycles →improve clock frequency, throughput
  - worsen latency (overhead of registers)
- Retiming
  - optimally distributing registers throughout a circuit → improve clock frequency
  - Reduce number of registers

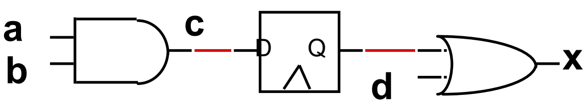




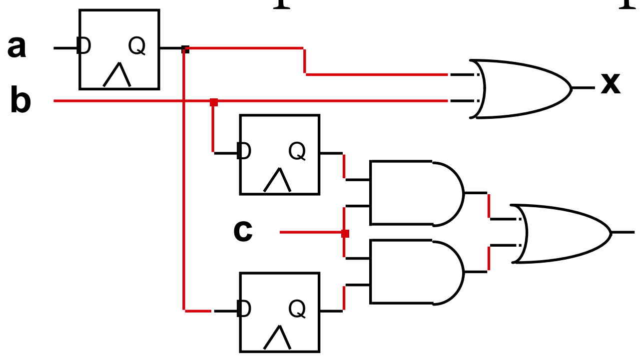
Retiming

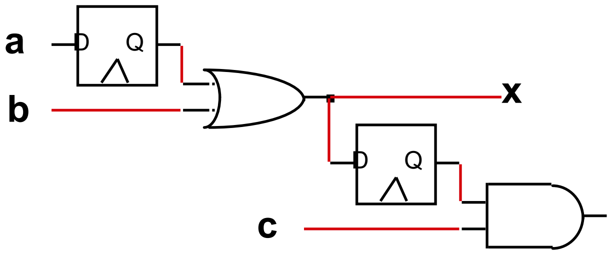
- Shortening critical paths





- Create simplification opportunities





- How?:
  - Move register(s) from input to outputs or vice-versa