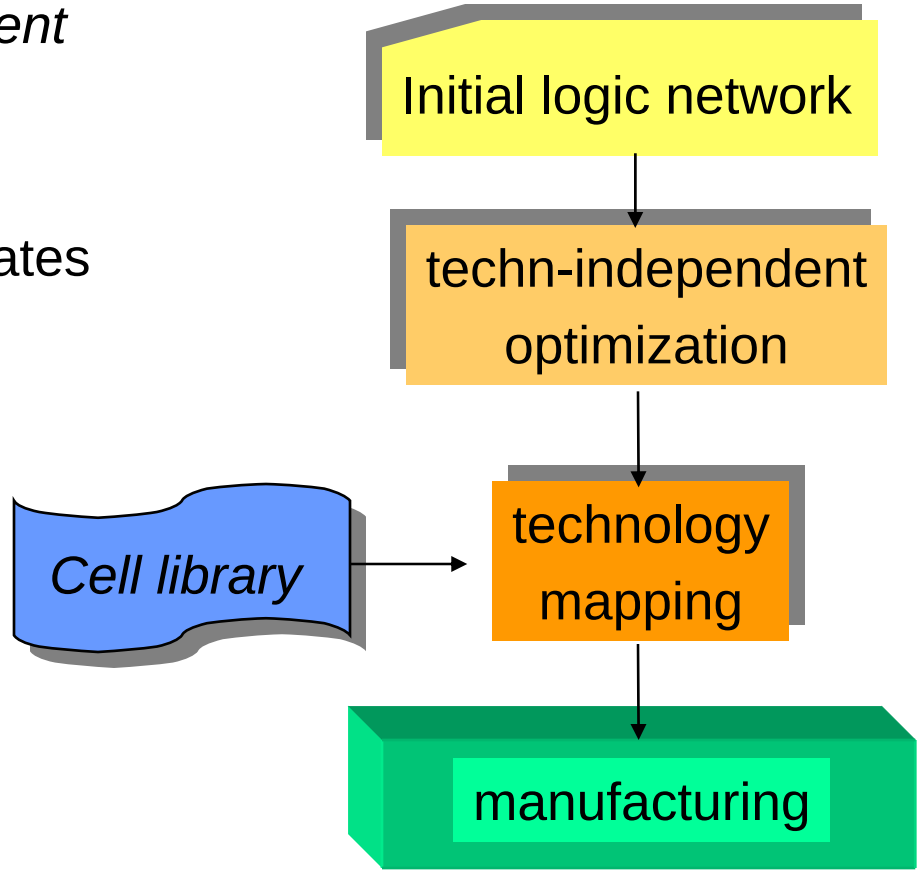
# Technology Mapping

*Slides adopted from A. Kuehlmann, UC Berkeley 2003*

# Technology Mapping

- Where is it used
  - After *technology independent* optimization
- Role
  - Assign logic functions to gates from custom library
  - Optimize for area, delay, power, etc.
- Also called ***library binding.***



# Technology-independent Optimization

- **Original logic network (16 literals):**

  $t_1 = a + bc$

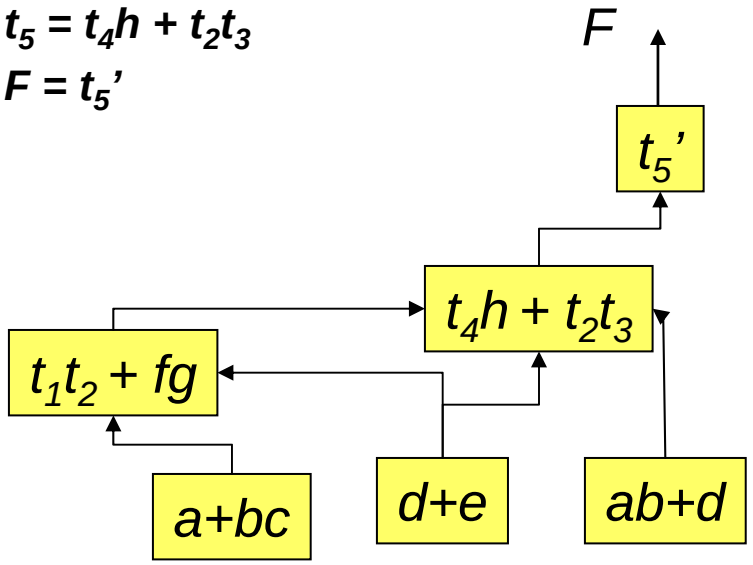
  $t_2 = d + e$

  $t_3 = ab + d$

  $t_4 = t_1t_2 + fg$

  $t_5 = t_4h + t_2t_3$

  $F = t_5'$




- ***Technology independent* optimization (14 literals):**

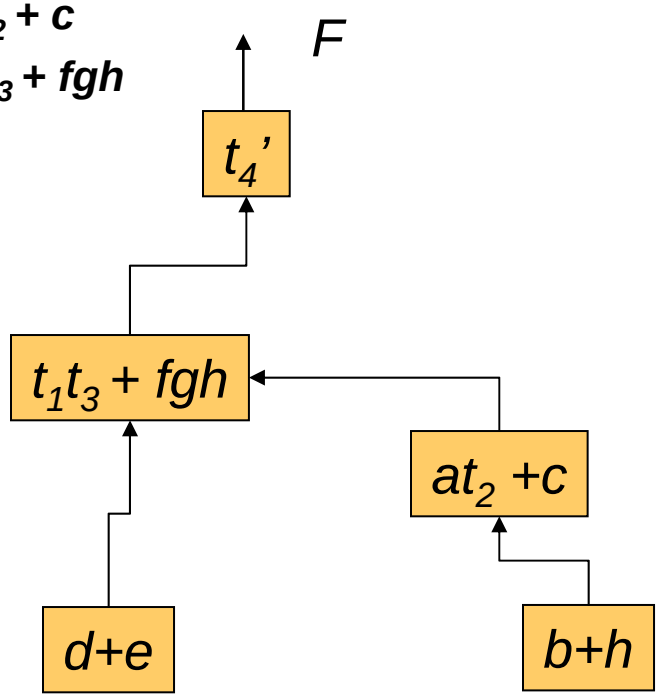
  $t_1 = d + e$

  $t_2 = b + h$

  $t_3 = at_2 + c$

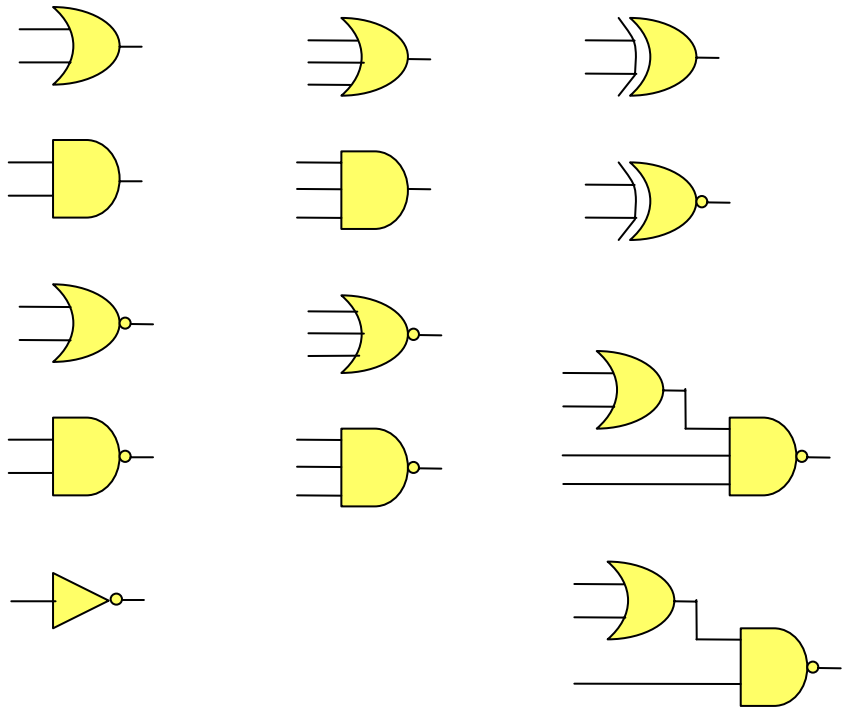
  $t_4 = t_1t_3 + fgh$

  $F = t_4'$



# Cell Library (library of gates)

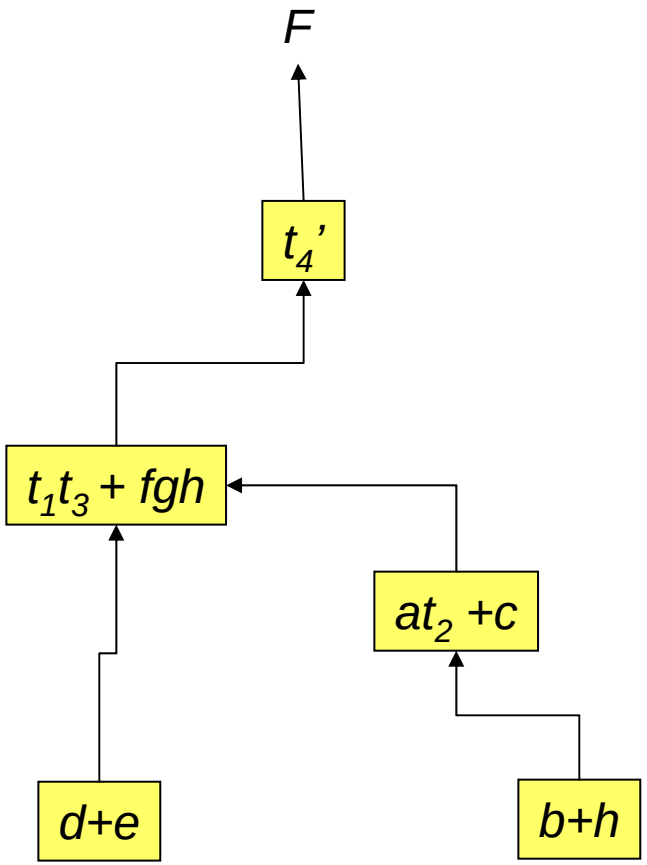
- Implement the optimized logic network using a set of ***gates*** which form a ***library***
- Each gate has a ***cost*** (area, delay, etc.)

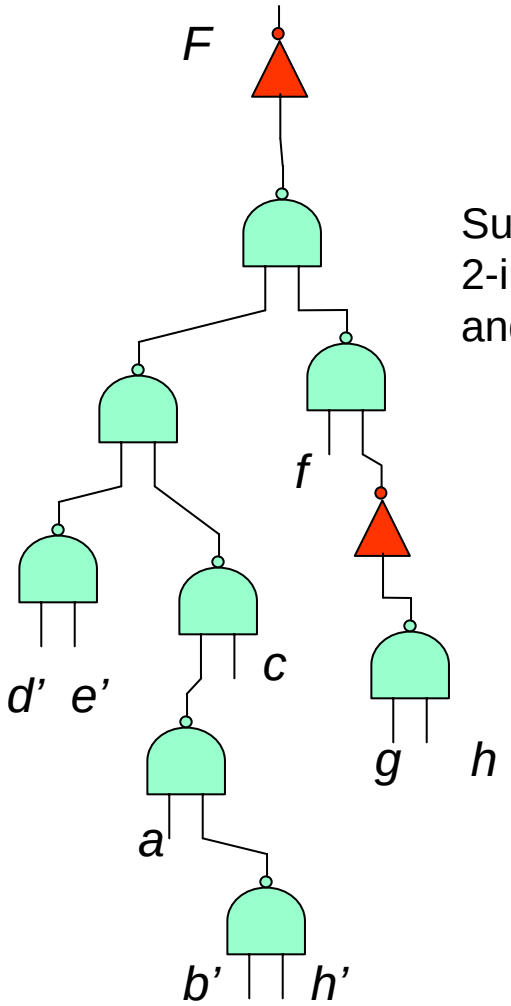


# Technology Mapping as Pattern Matching

- Represent each function of a network using a set of ***base functions***.
  - Typically the base is 2-input NANDs and inverters.
  - The set should be ***functionally complete.***
- This representation is called the ***subject graph***.
- Each gate of the library is likewise represented using the base set. This generates ***pattern graphs***.
  - Represent each gate in ***all possible*** ways.
- Cover the subject graph with patterns.
  - graph-based
  - binate covering

# Subject Graph




Subject graph of 2-input NANDs and invertors

# Pattern Matching Approach

- A ***cover*** is a collection of pattern graphs such that:
  - Every node of the subject graph is contained in one (or more) pattern graphs.
  - Each input required by a pattern graph is an output of some other graph.
- For minimum area, the cost of the cover is the sum of the areas of the gates in the cover.
- Technology mapping problem:
  - Find a ***minimum cost covering*** of the subject graph by choosing from the collection of pattern graphs for all the gates in the library.

# Subject Graph

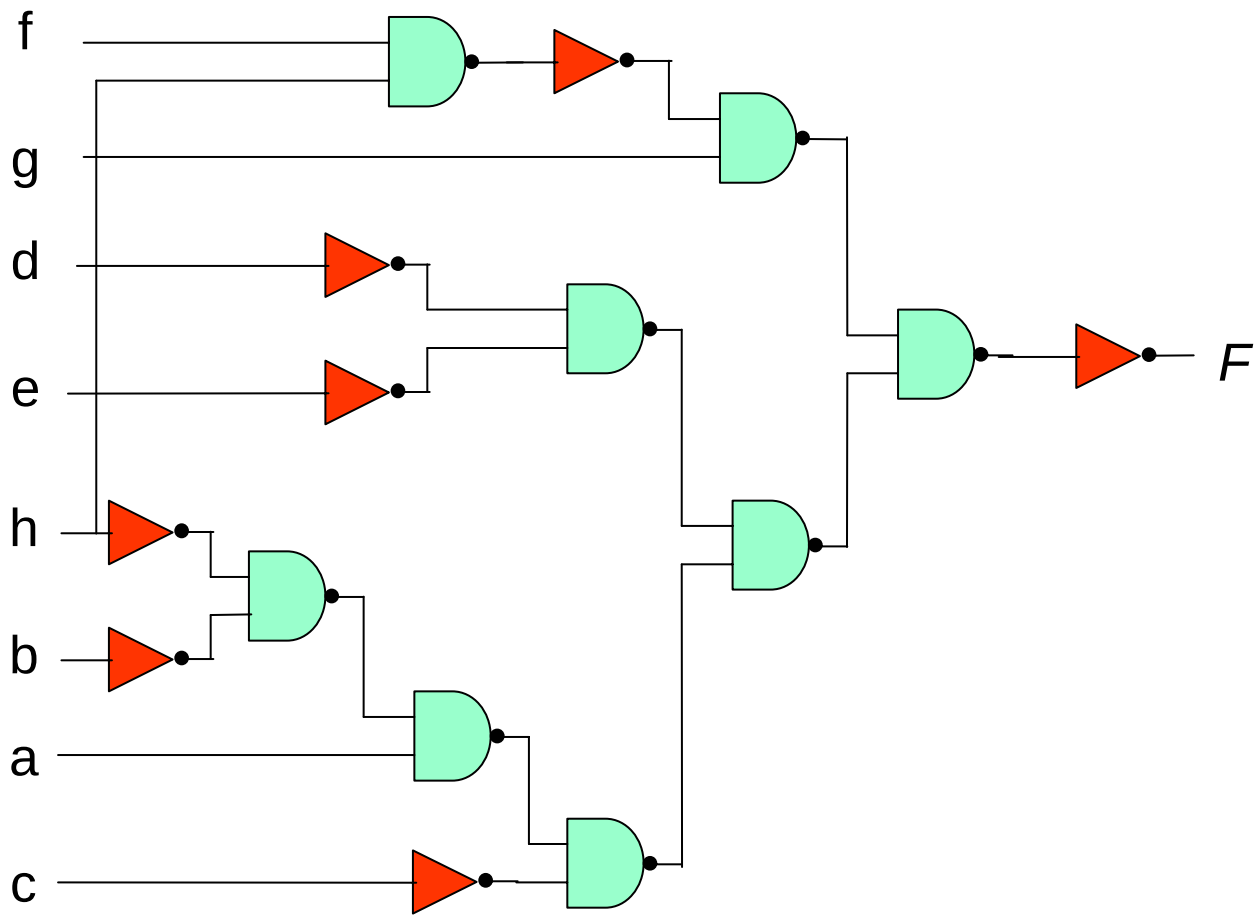
$t_1 = d + e$

$t_2 = b + h$

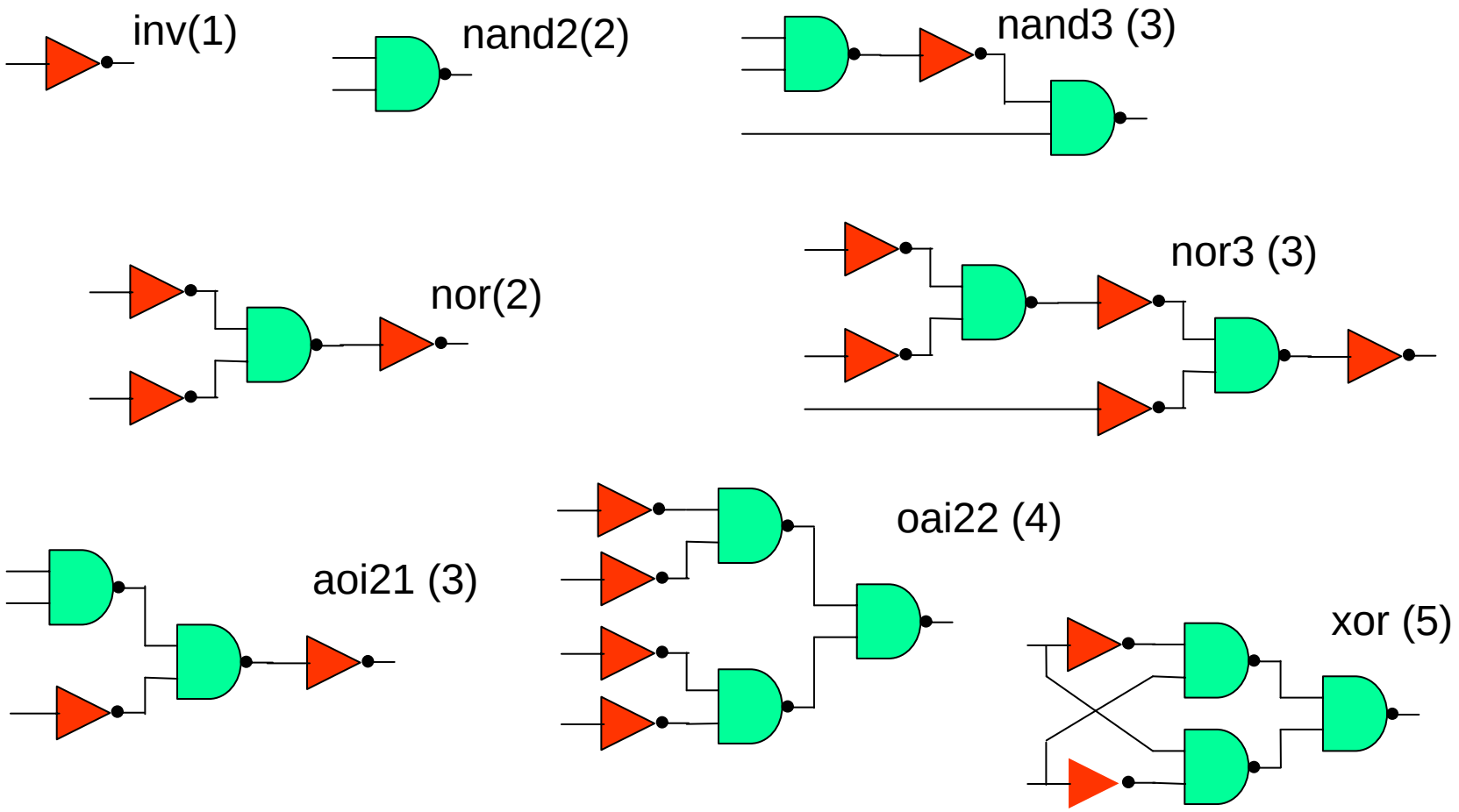
$t_3 = at_2 + c$

$t_4 = t_1t_3 + fgh$

$F = t_4'$



# Some Pattern Graphs from the Library




**Value in bracket indicates *cost***

# Subject graph covering (Trivial Covering)

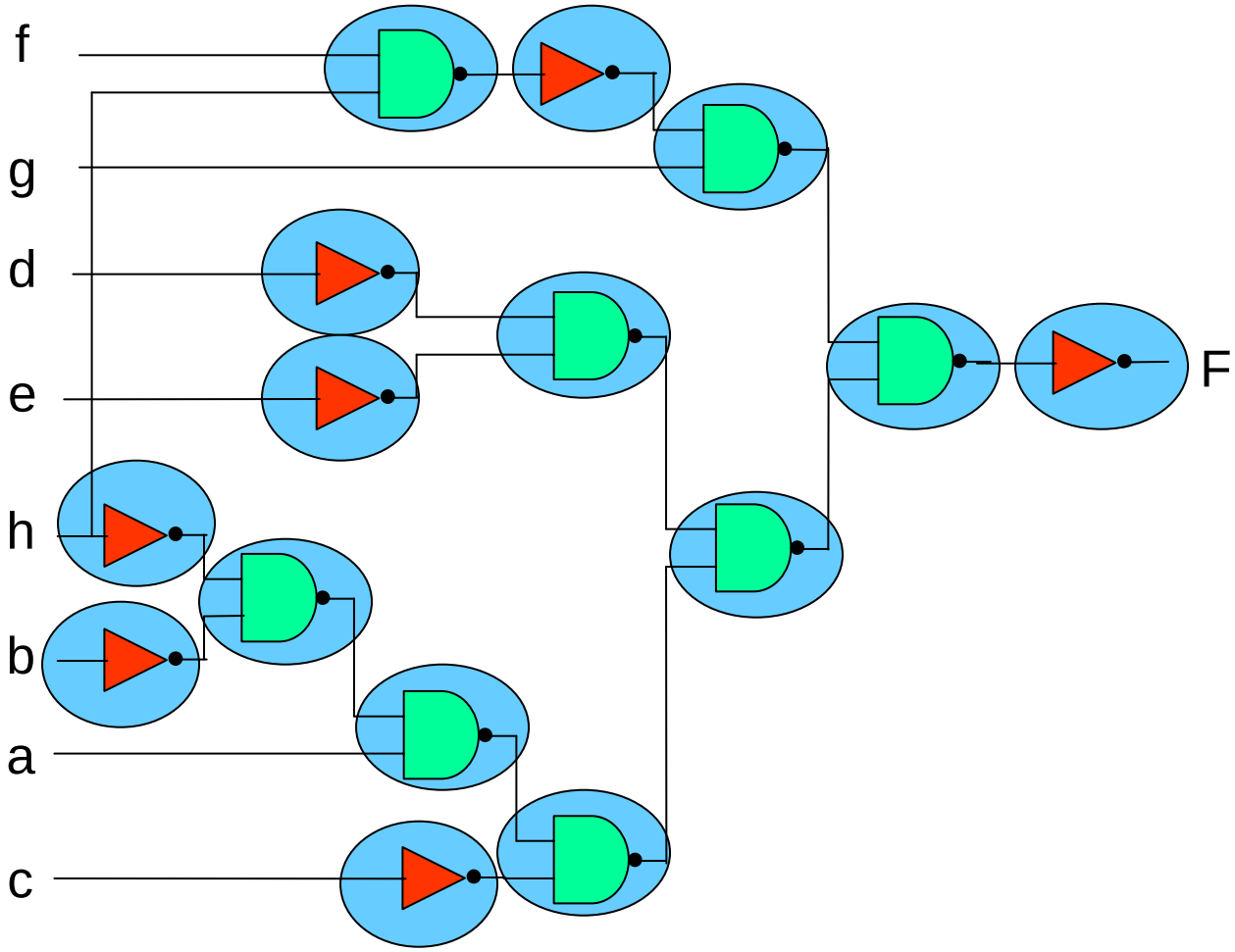
$t_1 = d + e$

$t_2 = b + h$

$t_3 = at_2 + c$

$t_4 = t_1t_3 + fgh$

$F = t_4'$




8 nand2, 7 inv

Total area: 23

# Better Covering

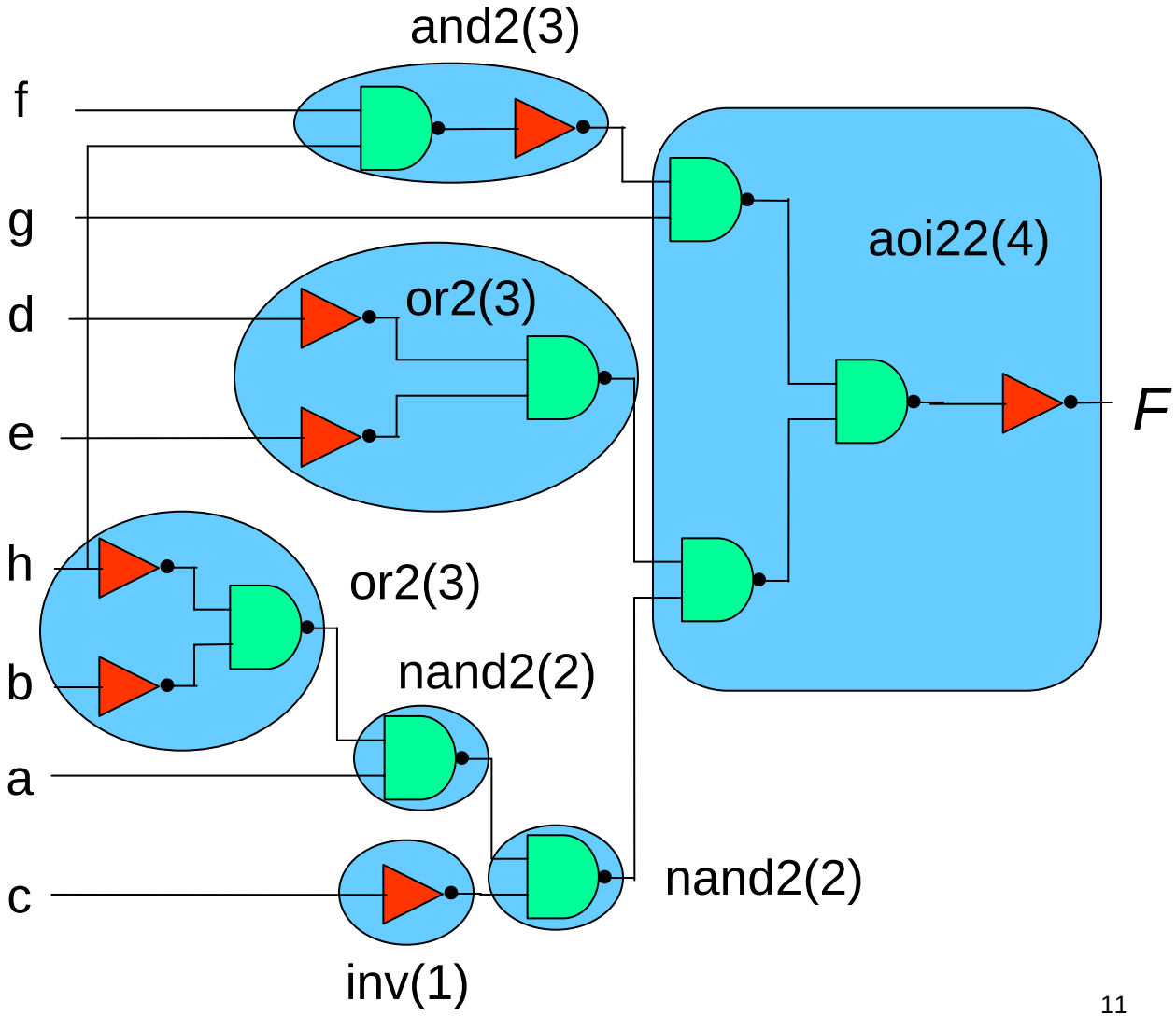
$t_1 = d + e$

$t_2 = b + h$

$t_3 = at_2 + c$

$t_4 = t_1t_3 + fgh$

$F = t_4'$




1 and2, 2 or2,
2 nand2, 1 inv,
1 aoi22

Total area: 18

# Alternate Covering

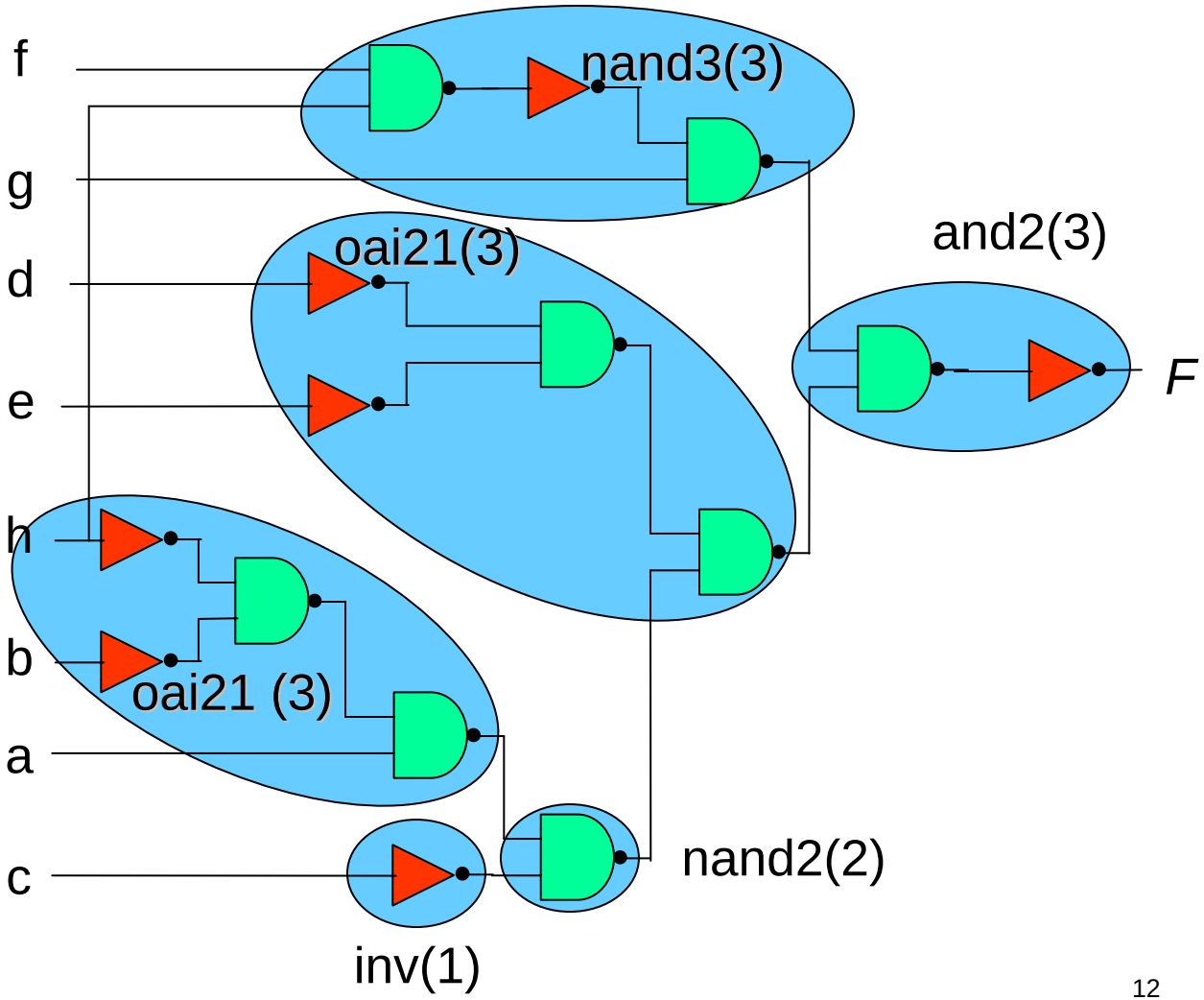
$t_1 = d + e$

$t_2 = b + h$

$t_3 = at_2 + c$

$t_4 = t_1t_3 + fgh$

$F = t_4'$




1 and2, 1 inv, 1 nand2, 1 nand3, 2 oai21

Total area: 15

# Tech. mapping using DAG Covering

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.

- Complexity of DAG covering:
  - NP-hard.
  - Remains NP-hard even when the nodes have out degree $\leq 2$
  - If subject DAG and pattern DAG's are trees, an efficient algorithm exists.
    - Using dynamic programming.
    - First proposed for optimal code generation in a compiler.

# DAG Covering Problem

- Compute all possible matches in the subject graph using the pattern graphs.
- Let $m_i \in \{0.1\}$ indicate the exclusion or the inclusion of match i in the cover.
  - $m_i$=0, if match i is not included
  - $m_i$=1, if match i is included
- Need to cover each node in the subject graph with a match.
  - Example: If node j is covered by matches $m_2$, $m_5$ and $m_{10}$, then the following clause must be satisfied to make sure that this node is covered.

    $(m_2 + m_5 + m_{10})$
  - Each node in the subject graph generates such a clause.
  - All the clauses must be satisfied to cover all the nodes in the subject graph.

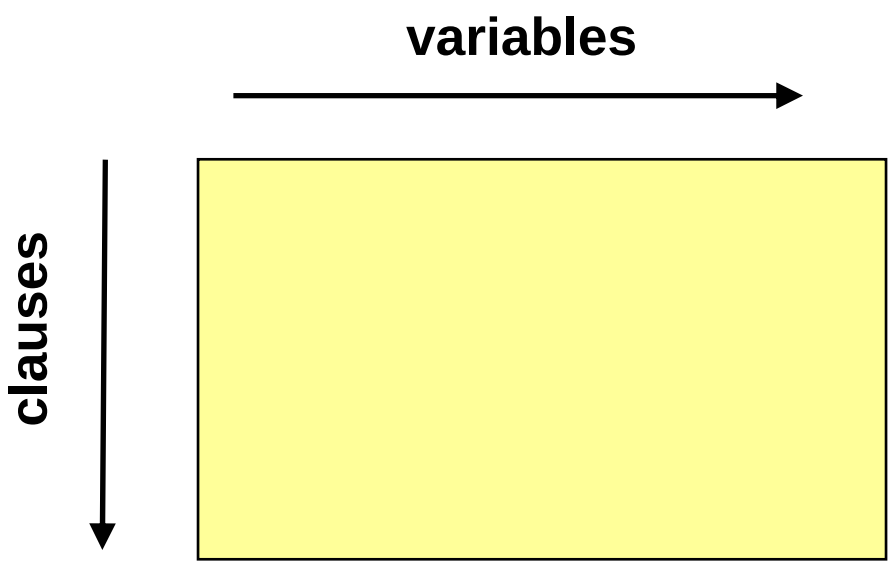
- Simply covering all the nodes in the subject graph is not enough.
  - Need to make sure that the inputs of the matches in the cover are also the outputs of some matches in the cover.
- Let $m_i$ have subject graph nodes $s_1$, $s_2$, …, $s_n$ as inputs.
  - If $m_i$ is in the cover, then for each $s_j$ that is not a primary input, there must be some match with $s_j$ an an output.
  - Let $S_j$ be the clause for all the matches that result in $s_j$ as an output.
    - Thus, $m_i \Rightarrow S_j$ for each $s_j$ that is not a primary input.
      - $m_i \Rightarrow S_j \equiv m_i' + S_j$
      - This ensures that the inputs to $m_i$ are outputs of some match (or primary inputs).

- Each primary input must have some match that generates it as an output (an $S_j$ type clause).
- Each clause must be satisfied.

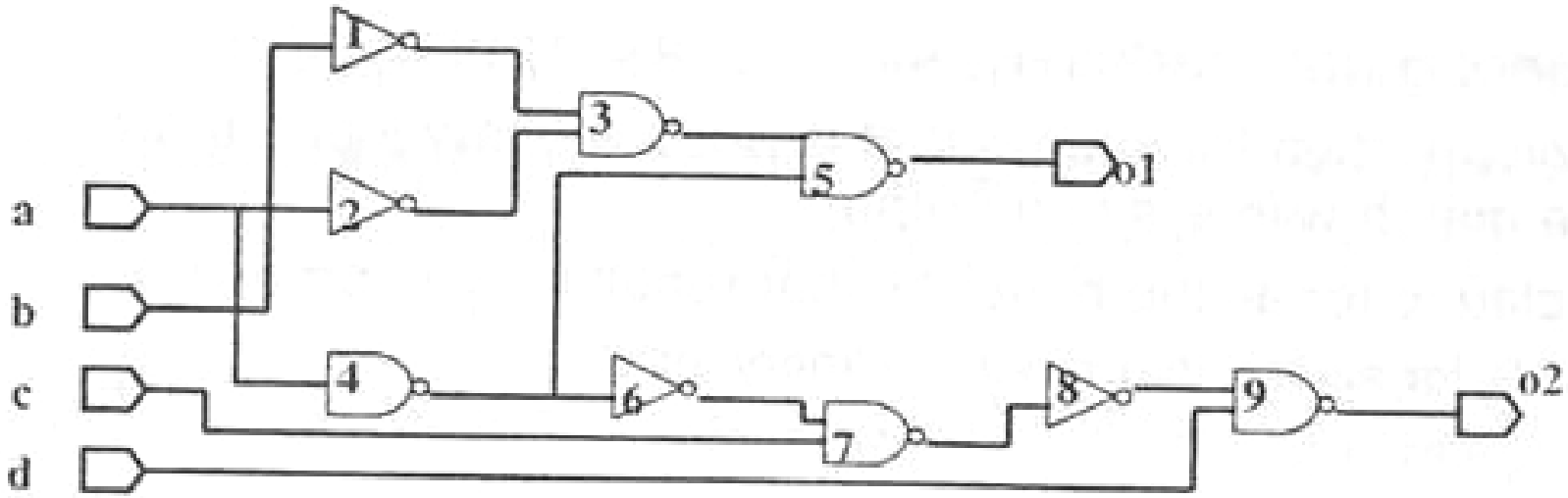
# DAG Covering as Binate Covering

- Satisfying each clause is equivalent to the binate covering problem.
  - Need to cover each clause.
  - Clause may be covered by a variable selected in positive or negative phase.
  - Need to find a minimum cost cover.
    - With area as the cost function:
      - $m_i = 0$, cost = 0
      - $m_i = 1$, cost = area cost of pattern graph in mi
    - Very hard when cost of a match is not independent of other matches.

# Covering Table



# Binate Covering Example



| Match | Gate | Cost | Inputs | Produces | Covers |
|---|---|---|---|---|---|
| m1 | inv | 1 | b | g1 | g1 |
| m2 | inv | 1 | a | g2 | g2 |
| m3 | nand2 | 2 | g1,g2 | g3 | g3 |
| m4 | nand2 | 2 | a,b | g4 | g4 |
| m5 | nand2 | 2 | g3,g4 | g5 | g5 |
| m6 | inv | 1 | g4 | g6 | g6 |
| m7 | nand2 | 2 | g6,c | g7 | g7 |
| m8 | inv | 1 | g7 | g8 | g8 |
| m9 | nand2 | 2 | g8,d | g9 | g9 |
| m10 | nand3 | 3 | g6,c,d | g9 | g7,g8,g9 |
| m11 | nand3 | 3 | a,b,c | g7 | g4,g6,g7 |
| m12 | xnor2 | 5 | a,b | g5 | g1,g2,g3,g4,g5 |
| m13 | nand4 | 4 | a,b,c,d | g9 | g4,g6,g7,g8,g9 |
| m14 | oai21 | 3 | a,b.g4 | g5 | g1,g2,g3,g5 |

- Each node must be covered by some match.

  (m1+m12+m14)(m2+m12+m14)(m3+m12+m14)(m4+m11+m12+m13)(m5+m12+m14)(m6+m11+m13)(m7+m10+m11+m13)(m8+m10+m13)(m9+m10+m13)

- Each selected match input must be a primary input or the output of some other selected match.

  (m3′+m1)(m3′+m2) (m5′+m3) (m5′+m4) (m6′+m4) (m7′+m6) (m8′+m7) (m9′+m8) (m10′+m6) (m14′+m4)

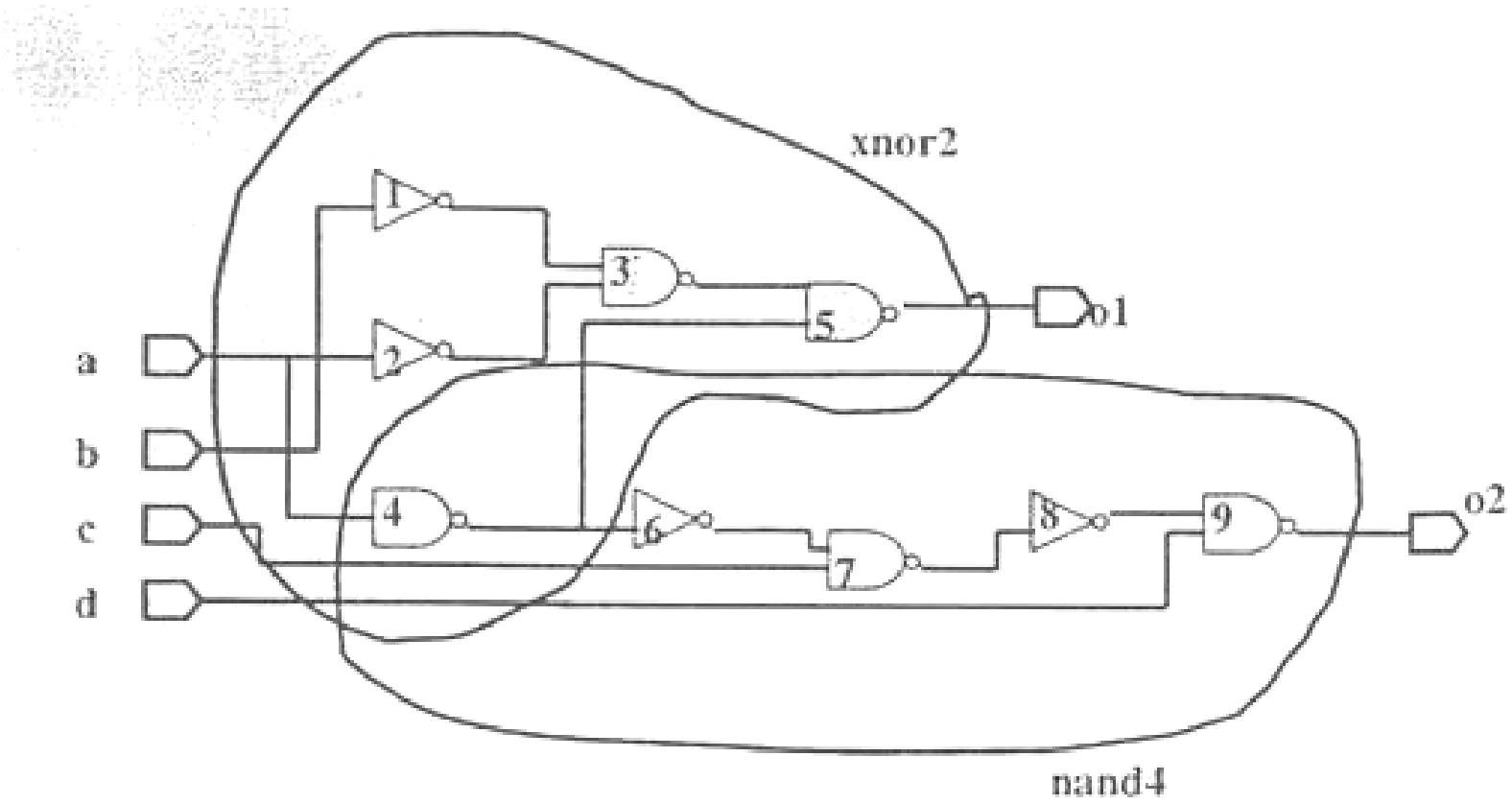
- The two primary outputs must be the outputs of selected matches.

  (m5+m12+m14)(m9+m10+m13)

- Covering expression has 58 prime implicants (i.e. 58 minimal solutions).
  - Least cost solution is:

    m3′m5′m6′m7′m8′m9′m10′m12m13m14′
  - Uses two gates for a cost of 9.
    - m12: xor
    - m13: nand4

# Final Solution



# Problems with Binate Covering

- Intrinsically a hard problem.
  - Search techniques seem to work well for large unate covering problems, but not for binate covering problems.
- Problem size is very large.
  - Large number of matches.
  - Large number of clauses (rows).
- Cost function is limited.
  - Cost for a match must be independent of the cost of other matches.
  - Works for ***area*** as a cost, but not for ***delay*** and ***power***.

# Dynamic Programming

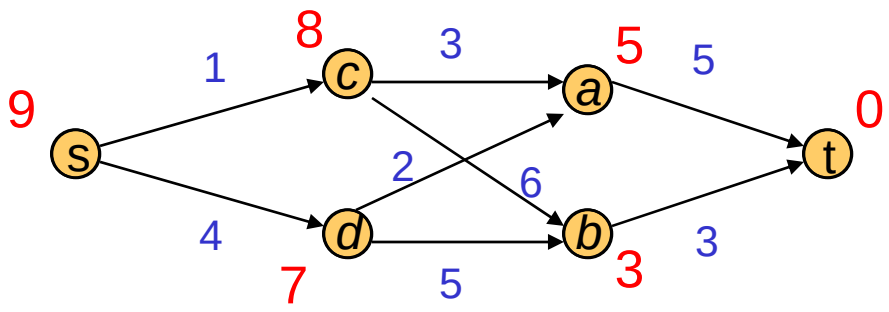
- An algorithmic method that solves an optimization problem by decomposing it into a *sequence of decisions*.
- Such decisions lead to an optimum solution if the following *Principle of Optimality* is satisfied [Bellman 1957]:
  - *An optimal sequence of decisions has the property that whatever the initial state and decisions are, the remaining decisions must constitute an optimal decision sequence w.r.to the state resulting from the fist decision*.
- Typical recursive equation:

$$cost(i) = min_k\{ d_{ik} + cost(k) \}$$

# Dynamic Programming - example

- Example: shortest path problem in a layered network

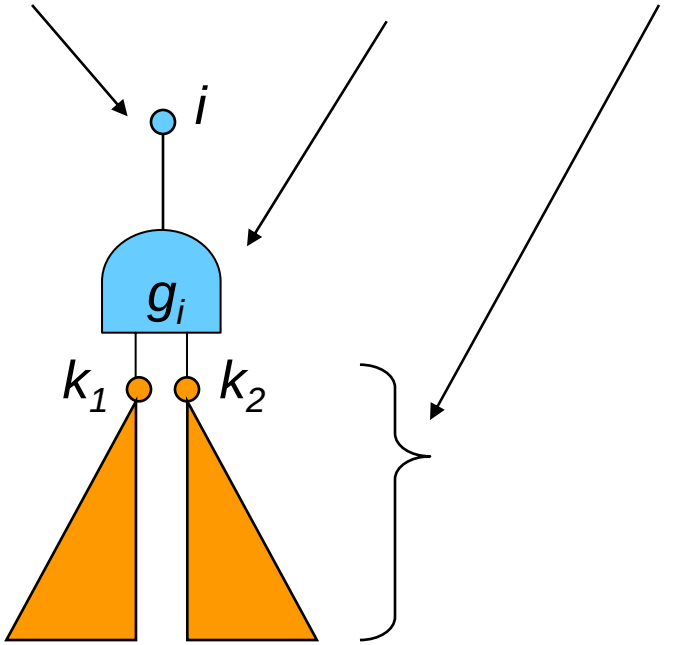
$$cost(i) = min_k\{\, d_{ik} + cost(k) \,\}$$



# Dynamic Programming - example

- Similarly, for node *i* at gate $g_i$:

$$cost(i) = min_k\{cost(g_i) + \sum_k cost(k)\}$$



# Optimal Tree Covering by Trees

- Partition subject graph into forest of trees
- Cover each tree optimally using *dynamic programming*

Given:

  - Subject trees (networks to be mapped)
  - Forest of patterns (gate library)
- Consider a node *N* of a subject tree
- Recursive Assumption: for all children of *N,* a best cost match (which implements the node) is known
- Cost of a leaf of the tree is 0.
- Compute cost of each pattern tree which matches at *N*,

  Cost = *SUM* of best costs of implementing each input of pattern

  plus the cost of the pattern
- Choose least cost matching pattern for implementing *N*

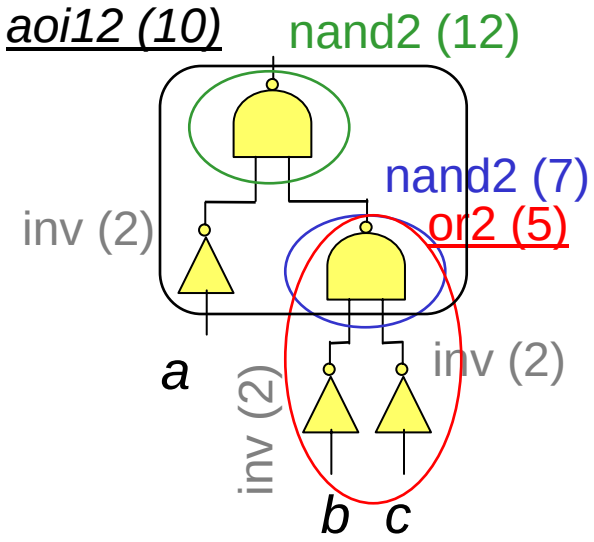
# Optimum Area Algorithm

```
Algorithm OPTIMAL_AREA_COVER(node) {
  foreach input of node {
    OPTIMAL_AREA_COVER(input);// satisfies recurs. assumption
   }
  // Using these, find the best cover at node
  node→area = INFINITY;
  node→match = 0;
  foreach match at node {
    area = match→area;
    foreach pin of match {
      area = area + pin→area;
    }
    if (area < node→area) {
      node→area = area;
      node→match = match;
    }
  }
}
```

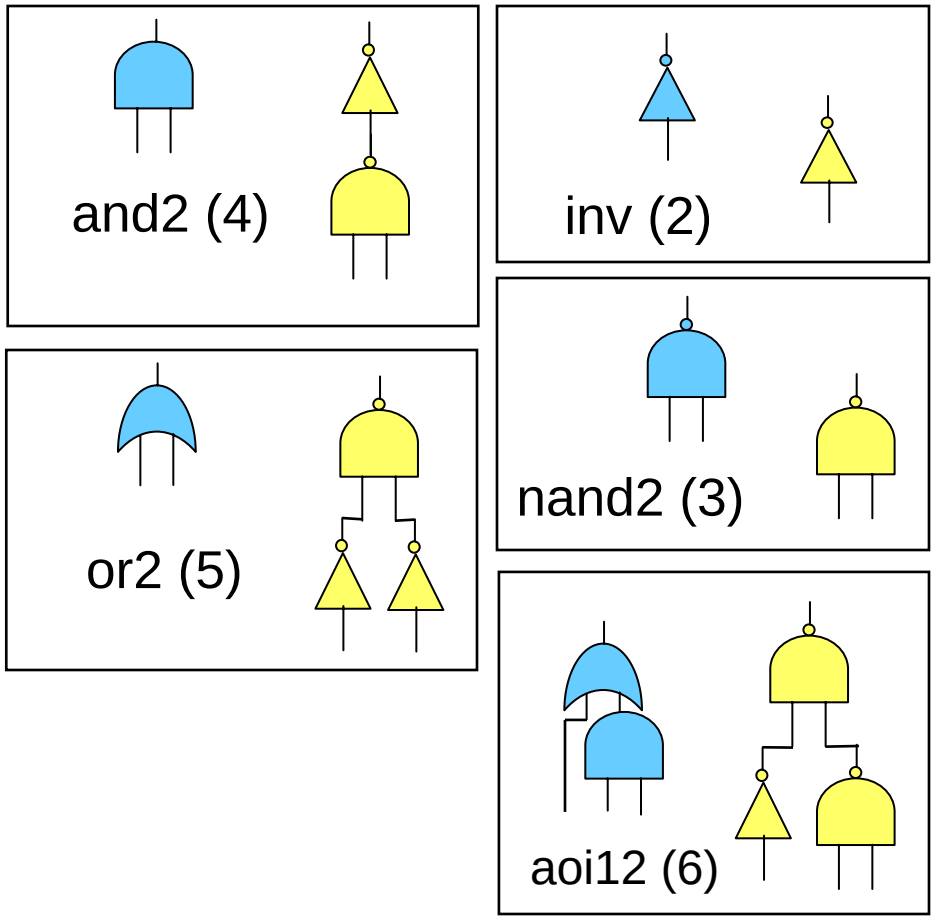
# Technology Mapping - example

- *F = (a' (b+c))' = a + b' c'*

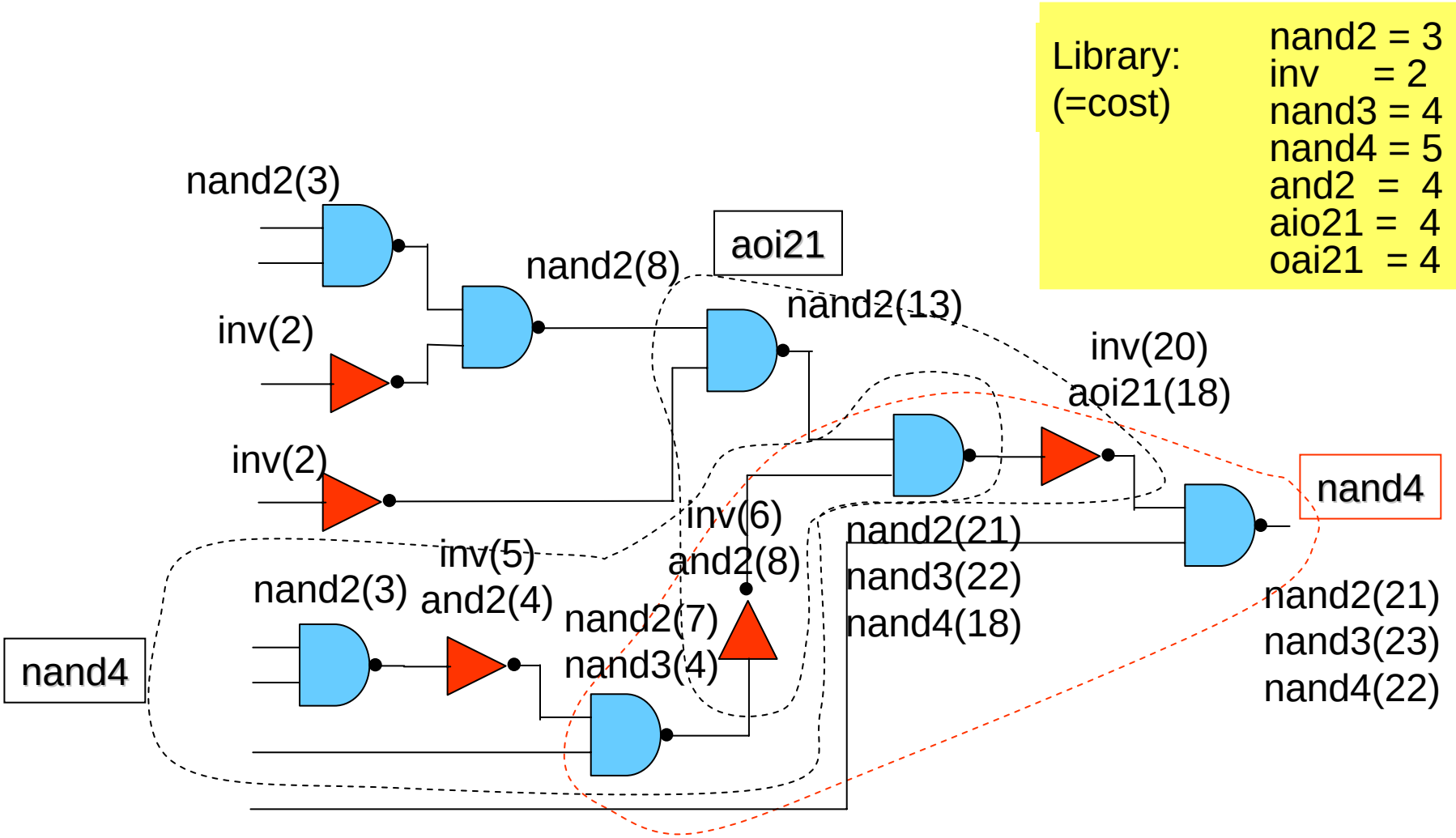
Subject tree (network)




Pattern trees (library)



# Tree Covering in Action



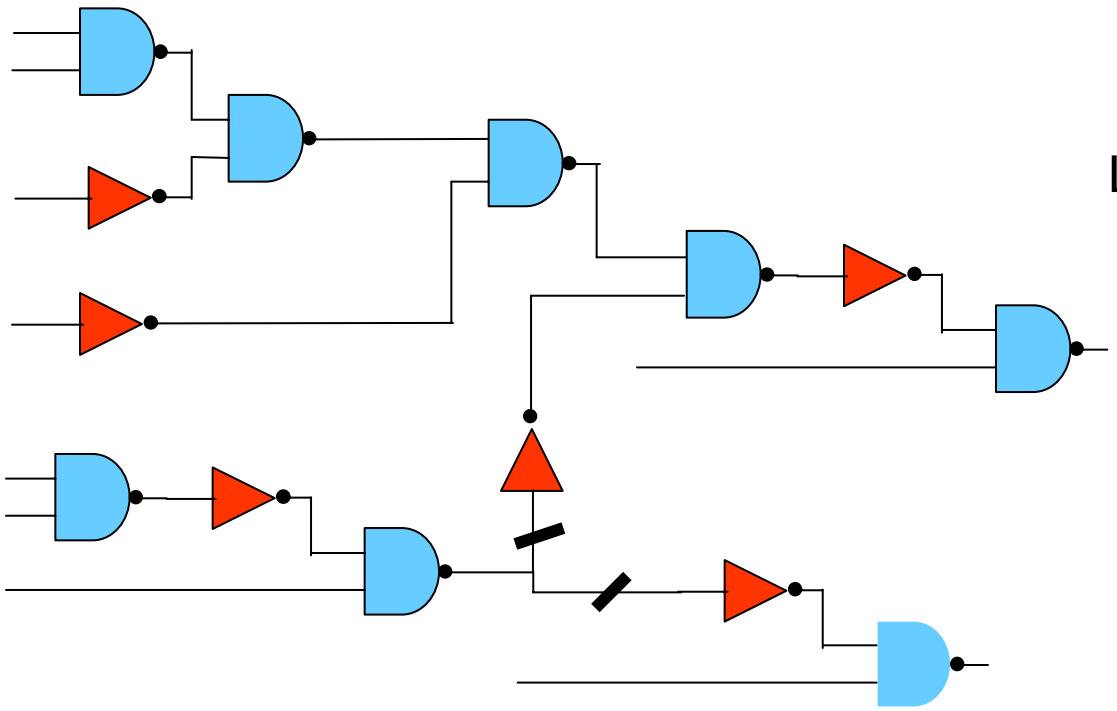
# Complexity of Tree Covering

- Complexity is controlled by finding *all* sub-trees of the subject graph which are isomorphic to a pattern tree.
- Complexity of covering is *linear* in both size of subject tree and size of collection of pattern trees
- But: for the overall mapping, must add complexity of *matching*
  - *Complexity = O(nodes)*(complexity of matching)*

# Partitioning the Subject DAG into Trees

Trivial partition: break the graph at all multiple-fanout points

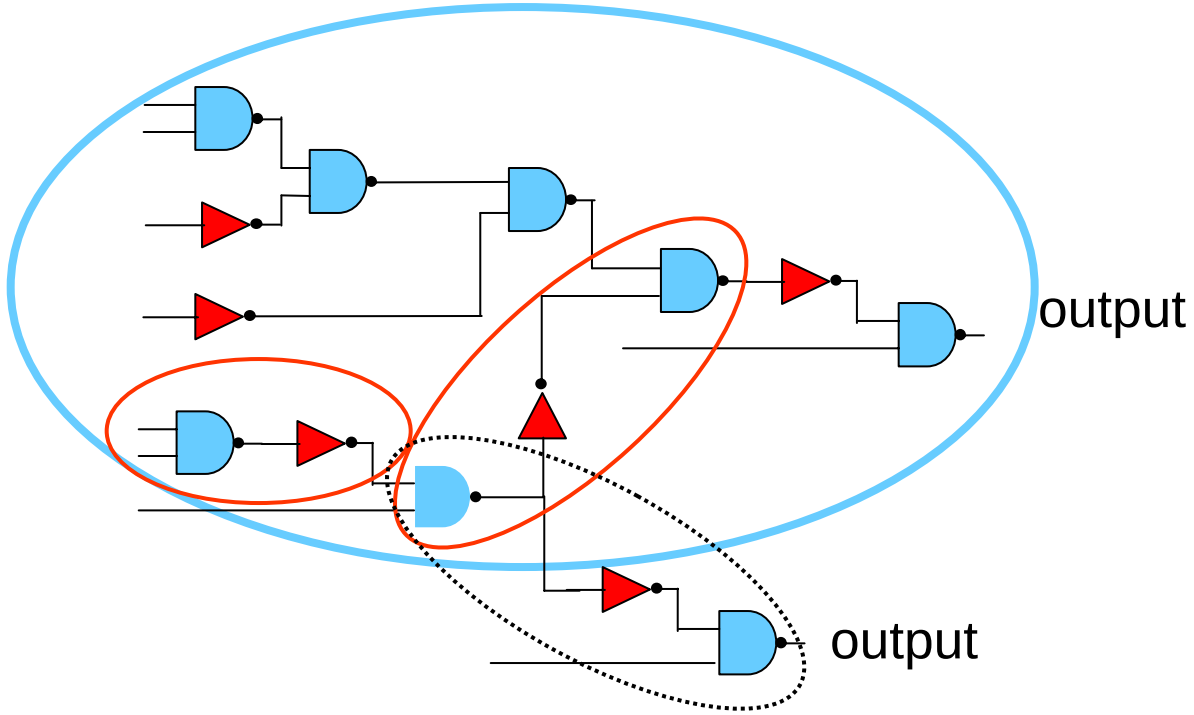
- leads to no “duplication” or “overlap” of patterns
- drawback - sometimes results in many of small trees



Leads to 3 *trees*

# Partitioning the subject DAG into trees

- Single-cone partition:
  - from a single output, form a large tree back to the primary inputs;
  - map successive outputs until they hit match output formed from mapping previous primary outputs.
    - Duplicates some logic (where trees overlap)
    - Produces much larger trees, potentially better area results



# Min-Delay Covering

- For trees:
  - identical to min-area covering
  - use optimal delay values within the dynamic programming paradigm
- For DAGs:
  - if delay does not depend on number of fanouts: use dynamic programming as presented for trees
  - leads to optimal solution in polynomial time
    - “we don’t care if we have to replicate logic”
- Combined objective
  - e.g. apply delay as first criteria, then area as second
  - combine with static timing analysis to focus on critical paths