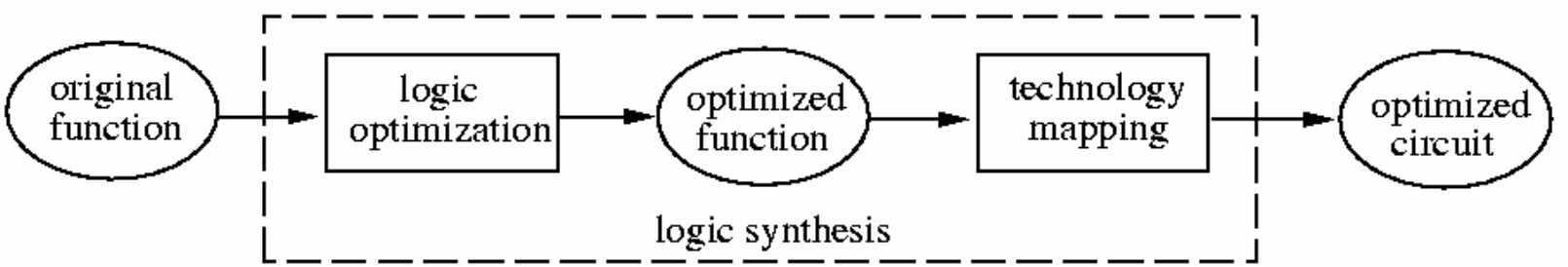
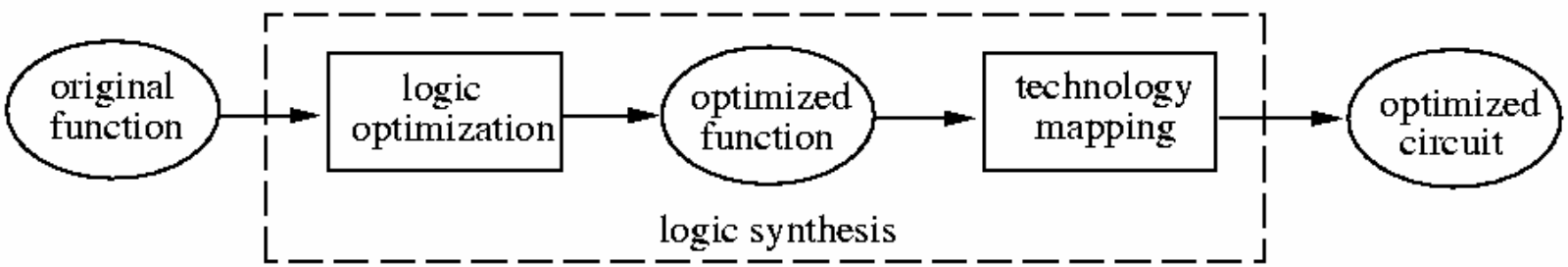
Unit 4: Formal Verification

- Course contents
  - Logic synthesis basics
  - Binary-decision diagram (BDD)
  - Verification
  - Logic optimization
  - Technology mapping
- Readings
  - Chapter 11



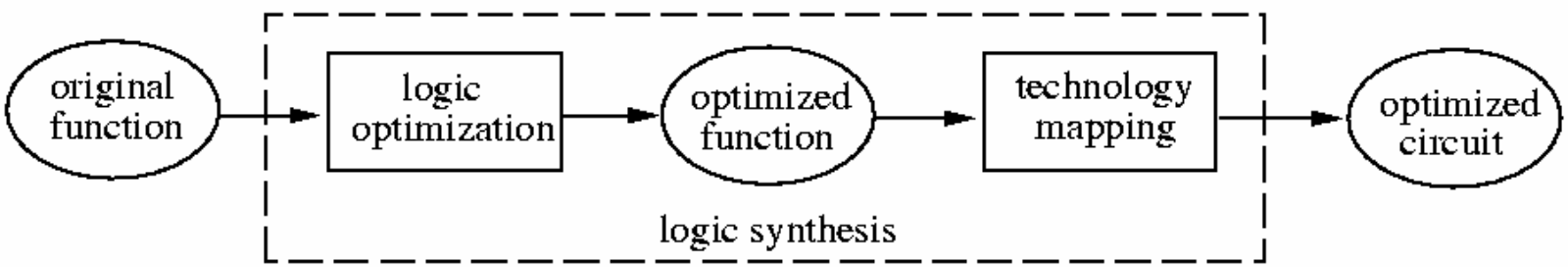
Logic Synthesis & Verification

- **Logic synthesis** programs transform Boolean expressions or register-transfer level (RTL) description (in Verilog/VHDL/C) into logic gate networks (netlist) in a particular library.
  - Three different tasks
    - two-level combinational synthesis
    - multilevel combinational synthesis
    - sequential synthesis
  - Optimization goals: minimize area, delay, and power, etc
- **Verification:** Checks the equivalence of a specification and an implementation.



Logic Synthesis & Verification

- **Technology-independent** optimization
  - Works on Boolean expression equivalent.
  - Estimates size based on # of literals.
  - Uses don't-cares, common factor extraction (factorization), etc. to optimize logic.
  - Uses simple delay models.
- **Technology-dependent** optimization: **technology mapping/library binding**
  - Maps Boolean expressions into a particular cell library.
  - May perform some optimizations in addition to simple mapping.
  - Uses more accurate delay models based on cell structures.

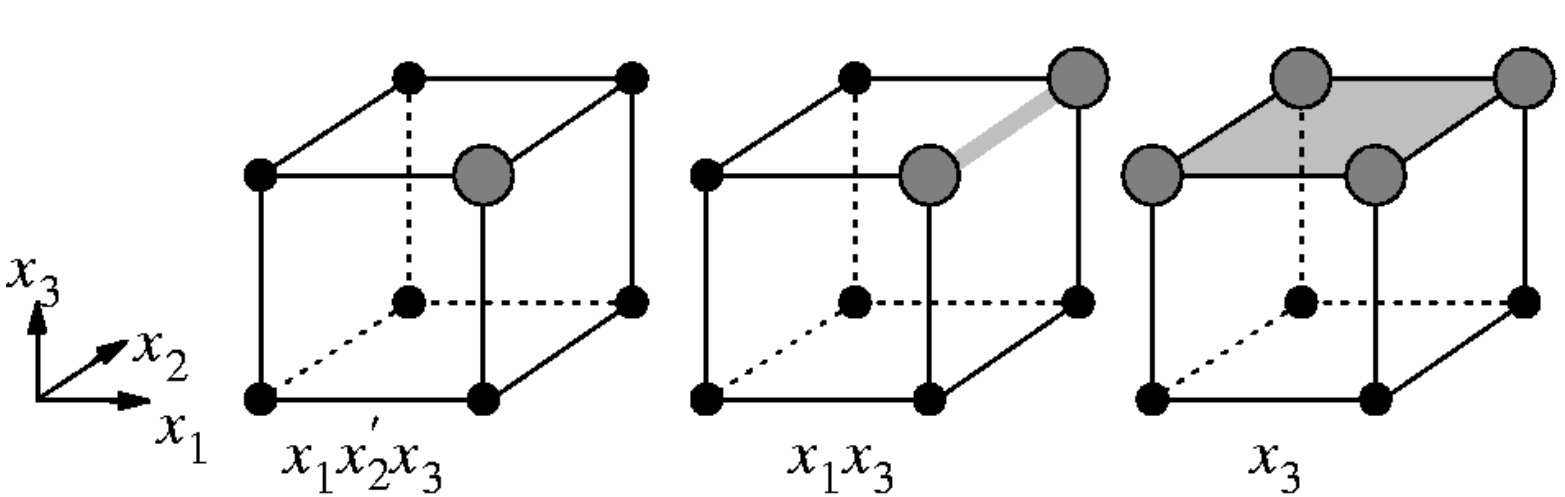


Boolean Functions

- $B = \{0,1\}$, $Y = \{0,1,D\}$
- A Boolean function f: $B^m \rightarrow Y^n$
  - $f = \overline{x}_1 \overline{x}_2 + \overline{x}_1 \overline{x}_3 + \overline{x}_2 x_3 + x_1 x_2 + x_2 \overline{x}_3 + x_1 x_3$
- Input variables: x_1, x_2, …
- The value of the output partitions B^m into three sets
  - the on-set
  - the off-set
  - the dc-set (don't-care set)

Minterms and Cubes

- A minterm is a product of all input variables or their negations.
  - A minterm corresponds to a single point in B^n.
- A cube is a product of the input variables or their negations.
  - The fewer the number of variables in the product, the bigger the space covered by the cube.

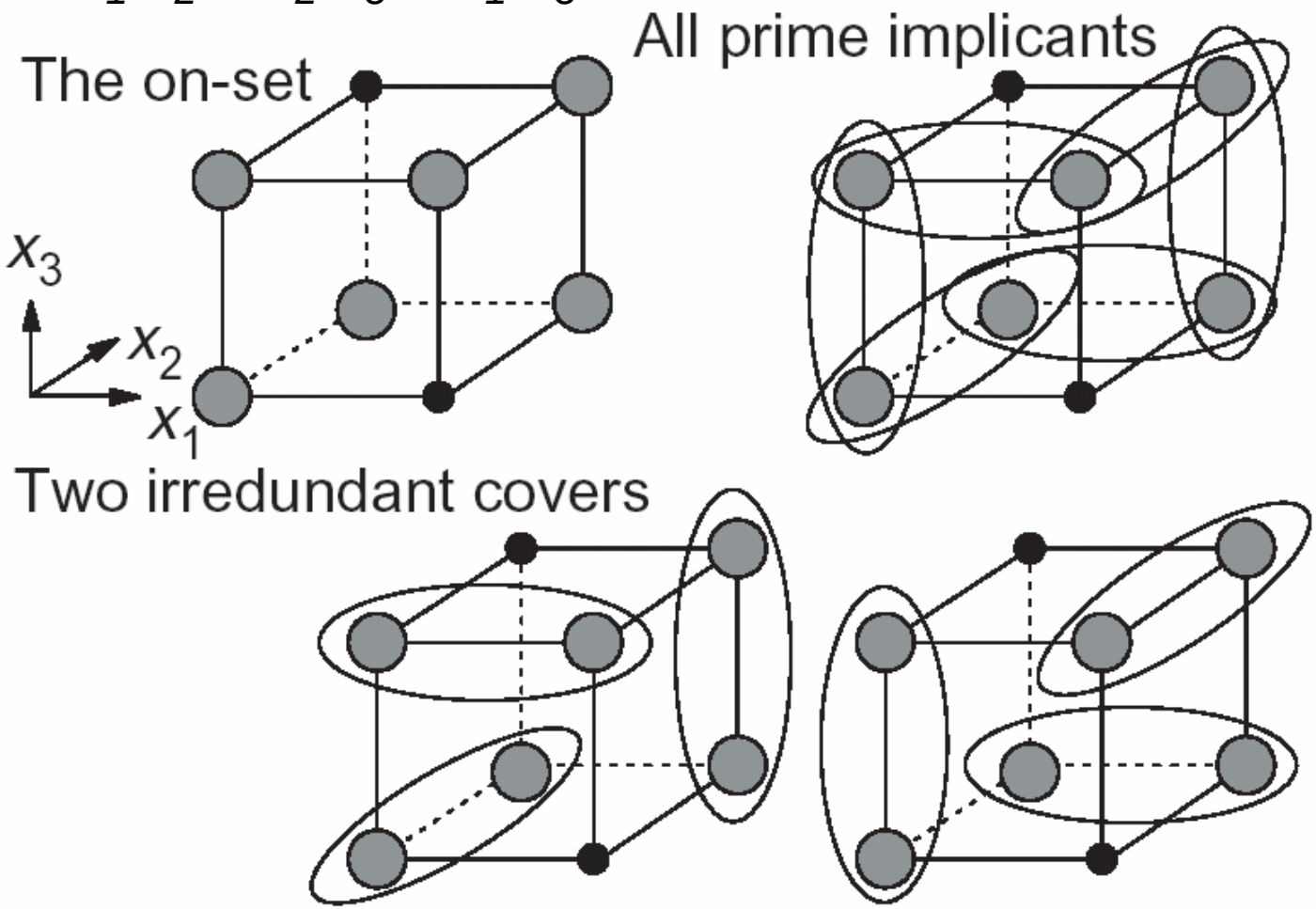


Implicant and Cover

- An **implicant** is a cube whose points are either in the on-set or the dc-set.
- A **prime implicant** is an implicant that is not included in any other implicant.
- A set of prime implicants that together cover all points in the on-set (and some or all points of the dc-set) is called a prime cover.
- A prime cover is irredundant when none of its prime implicants can be removed from the cover.
- An irredundant prime cover is minimal when the cover has the minimal number of prime implicants.

Cover Examples

- $f = \overline{x_1}\,\overline{x_3} + \overline{x_2}\,x_3 + x_1\,x_2$
- $f = \overline{x_1}\,\overline{x_2} + x_2\,\overline{x_3} + x_1\,x_3$



Canonical Forms

- A **canonical form** of a Boolean function is a unique representation of the function.
  - It can be used for verification purposes.
- The **truth table** or the **sum of minterms** are canonical forms
  - They grow exponentially with the number of input variables.
- A prime irredundant cover is not a canonical form.
- **Reduced ordered binary decision diagram (ROBDD):** a canonical form that is interesting from a practical point of view.

Logic Synthesis in Practice

- Specify the logic-level behavioral description of the circuit in some hardware-description language.
- Extract from this description the Boolean expressions related to the logic and represent them in some suitable internal form.
- Manipulate these expressions to obtain an optimized representation (two-level or multilevel).
- Perform **technology mapping**, a mapping from the abstract optimized representation to a netlist of cells from a library.

Binary-Decision Diagram (BDD) Principles

- **Restriction** resulting in the **positive** and **negative** cofactors of a Boolean function:

$$f_{x_i} = f(x_1, \ldots, x_{i-1}, '1', x_{i+1}, \ldots, x_m)$$
$$f_{\overline{x_i}} = f(x_1, \ldots, x_{i-1}, '0', x_{i+1}, \ldots, x_m)$$

  - $f = \overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}\,x_2\,\overline{x_3} + \overline{x_1}\,\overline{x_2}\,x_3 + x_1\,\overline{x_2}\,x_3 + x_1\,x_2\,\overline{x_3} + x_1\,x_2\,x_3$
    $f_{x1} = \overline{x_2}\,x_3 + x_2\,\overline{x_3} + x_2\,x_3$
    $f_{\overline{x1}} = \overline{x_2}\,\overline{x_3} + x_2\,\overline{x_3} + \overline{x_2}\,x_3$

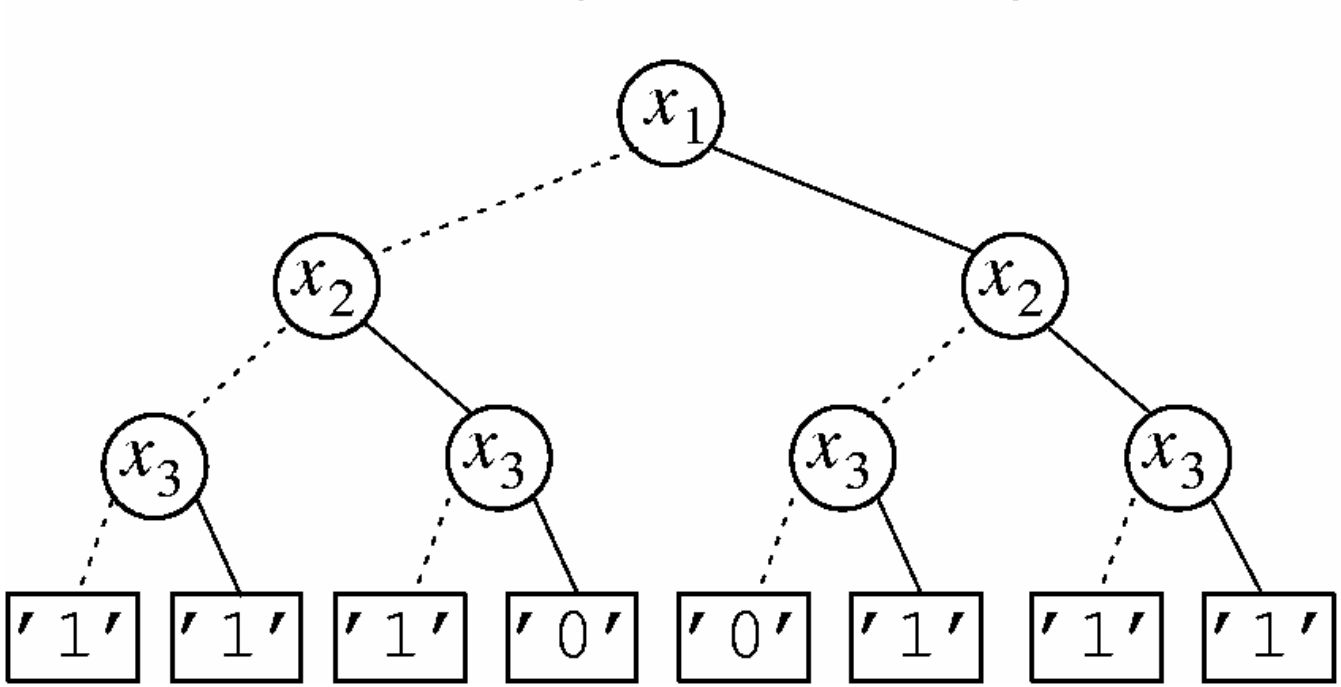
- **Shannon expansion** (already known to Boole) states:

$$f = x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}$$

- A complete expansion can be obtained by successively applying Shannon expansion on all variables of a function until either of the constant functions ’0’ or ’1’ are reached.

Example Ordered Binary-Decision Diagram (OBDD)

- The complete Shannon expansion can be visualized as a tree (solid lines correspond to the positive cofactors and dashed lines to negative cofactors).

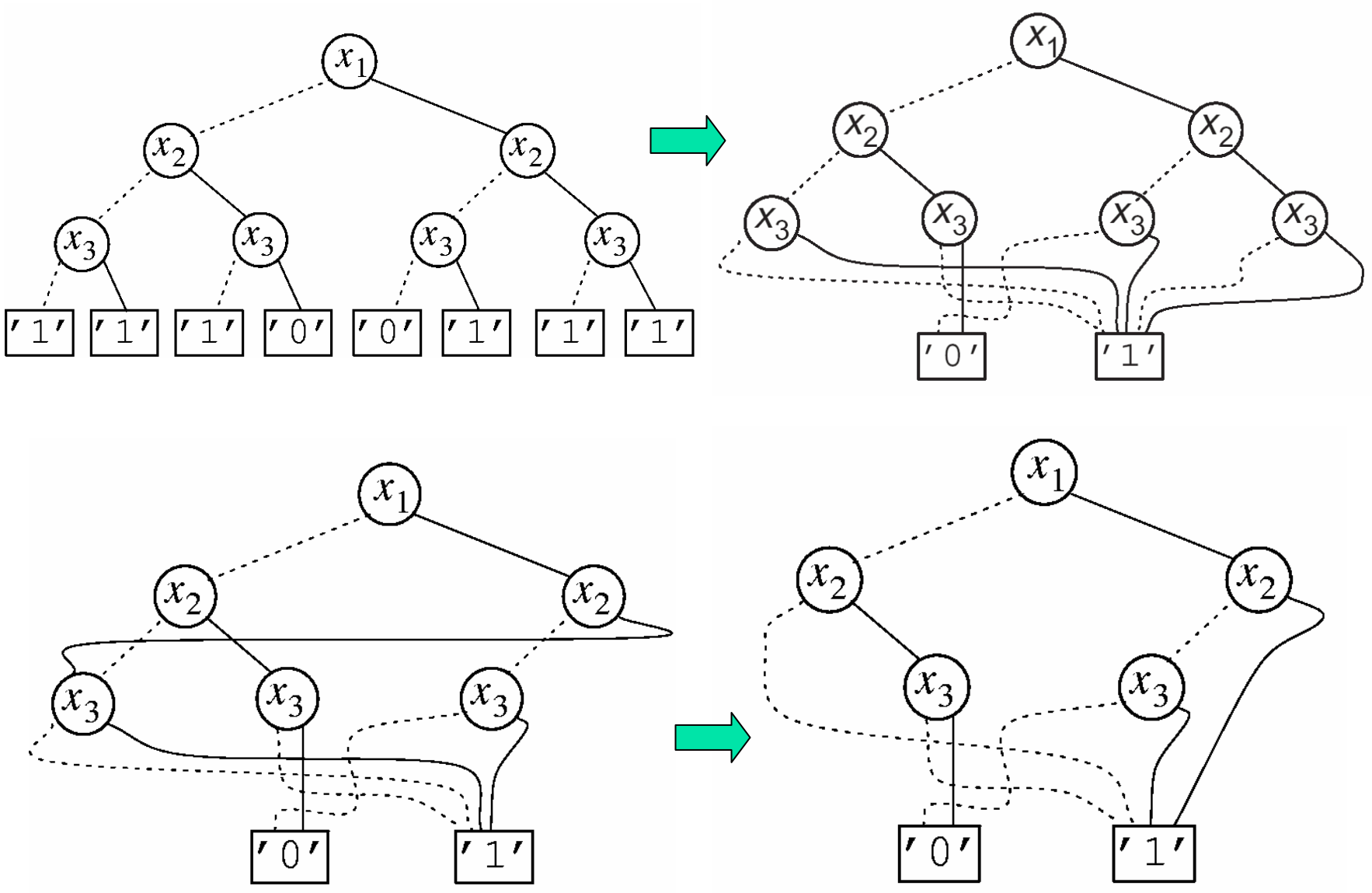


$$f = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3$$

Creating A Reduced OBDD (ROBDD)

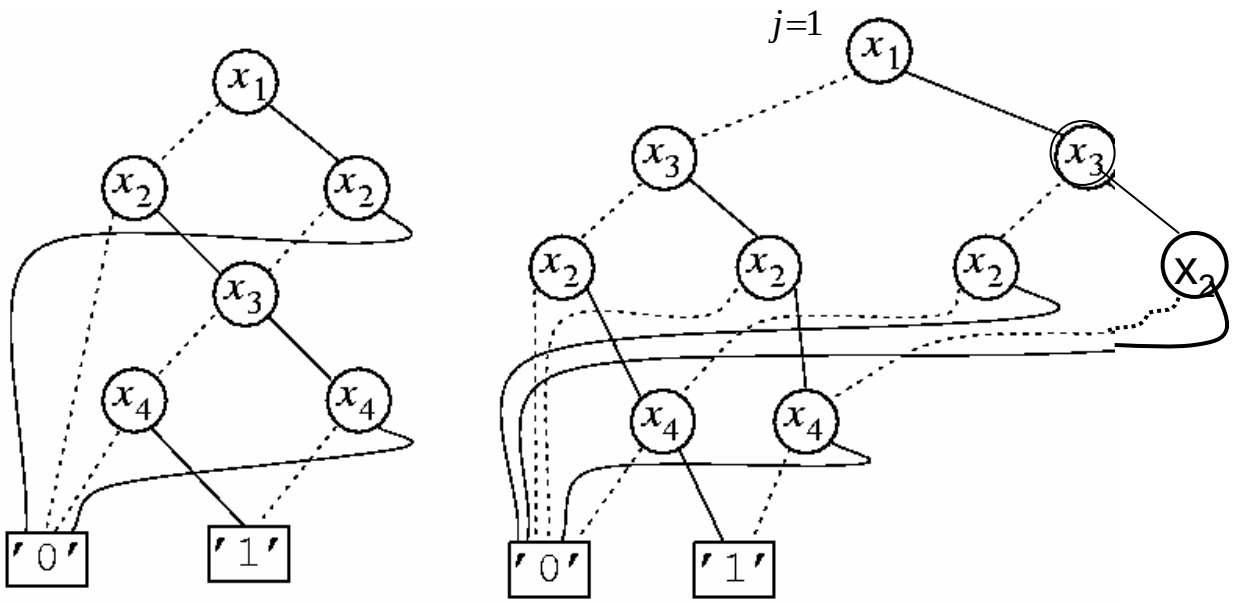
- An OBDD is a directed tree $G(V,E)$.
- Each vertex $v \in V$ is characterized by an associated variable $\phi(v)$, a high subtree $\eta(v)$ (high(v)) and a *low* subtree $\lambda(v)$ (low(v)).
- Procedure to reduce an OBDD:
  - Merge all identical leaf vertices and appropriately redirect their incoming edges;
  - Proceed from bottom to top, process all vertices: if two vertices u and v are found for which $\phi(u) = \phi(v)$, $\eta(u) = \eta(v)$, and $\lambda(u) = \lambda(v)$, merge u and v and redirect incoming edges;
  - For vertices v for which $\eta(v) = \lambda(v)$, remove v and redirect its incoming edges to $\eta(v)$.

Reduction Example



ROBDD Properties

- The ROBDD is a canonical representation, given a fixed ordering of the variables.
- The ROBDD is a compact representation for many Boolean functions used in practice.
- Variable ordering can greatly affect the size of an ROBDD.
  - E.g., the parity function of *k* bits: $f = \prod_{j=1}^{k} x_{2j-1} \oplus x_{2j}$



A BDD Package

- A BDD package refers to a software program that can manipulate ROBDDs. It has the following properties:
  - Interaction with BDDs takes place through an abstract data type (functionality is independent from the internal representation used).
  - It supports the conversion of some external representation of a Boolean function to the internal ROBDD representation.
  - It can store multiple Boolean functions, sharing all vertices that can be shared.
  - It can create new functions by combining existing ones (e.g., $h = f \bullet g$).
  - It can convert the internal representation back to an external one.

BDD Data Structures

- A triple (ϕ,η,λ) uniquely identifies an ROBDD vertex.

```
struct vertex {
  char *φ;
  struct vertex *η, *λ;
  ...
}
```

- A unique table (implemented by a hash table) that stores all triples already processed.

```
struct vertex *old_or_new(char *φ, struct vertex *η, *λ)
{
  if ("a vertex v = (φ, η, λ) exists")
    return v;
  else {
    v ← "new vertex pointing at (φ, η, λ)";
    return v;
  }
}
```

Building an ROBDD

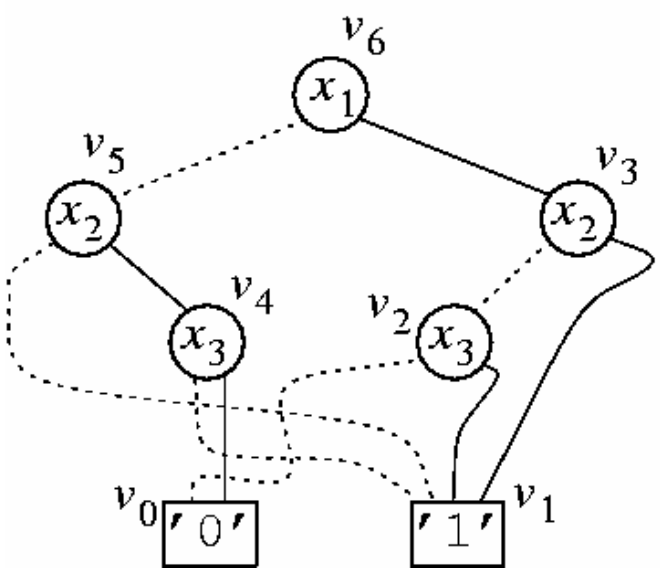
```
struct vertex *robdd_build(struct expr f, int i)
{
  struct vertex *η, *λ;
  struct char *φ;

  if (equal(f, '0'))
    return v0;
  else if (equal(f, '1'))
    return v1;
  else {
    φ ← π(i);
    η ← robdd_build(fφ, i + 1);
    λ ← robdd_build(f¬φ, i + 1);
    if (η = λ)
      return η;
    else
      return old_or_new(φ, η, λ);
  }
}
```

- The procedure directly builds the compact ROBDD structure.
- A simple symbolic computation system is assumed for the derivation of the cofactors.
- $\pi(i)$ gives the i^{th} variable from the top

robdd_build Example

robdd_build($\overline{x_1} \cdot \overline{x_3} + \overline{x_2} \cdot x_3 + x_1 \cdot x_2$, 1)

- ↰ robdd_build($\overline{x_2} \cdot x_3 + x_2$, 2)
  - ↰ robdd_build('1', 3)
    - v_1
  - ↳ robdd_build(x_3, 3)
    - ↰ robdd_build('1', 4)
      - v_1
    - ↳ robdd_build('0', 4)
      - v_0
    - $v_2 = (x_3, v_1, v_0)$
  - $v_3 = (x_2, v_1, v_2)$



↳ robdd_build($\overline{x_3} + \overline{x_2} \cdot x_3$, 2)

- ↰ robdd_build($\overline{x_3}$, 3)
  - ↰ robdd_build('0', 4)
    - v_0
  - ↳ robdd_build('1', 4)
    - v_1
  - $v_4 = (x_3, v_0, v_1)$
- ↳ robdd_build($\overline{x_3} + x_3$, 3)
  - ↰ robdd_build('1', 4)
    - v_1
  - ↳ robdd_build('1', 4)
    - v_1
  - v_1
- $v_5 = (x_2, v_4, v_1)$

$v_6 = (x_1, v_3, v_5)$

ROBDD Manipulation

- Separate algorithms could be designed for each separate operator on ROBDDs, such as AND, NOR, etc.
- However, the universal **if-then-else** operator '*ite*' is sufficient. *z* **=** *ite*(*f*,*g*,*h*), *z* equals *g* when *f* is true and equals *h* otherwise: $z = ite(f,g,h) = f \cdot g + \bar{f} \cdot h$
- Examples: $z = f \cdot g = ite(f,g,'0')$

  $z = f + g = ite(f,'1',g)$
- The *ite* operator is well-suited for a recursive algorithm based on ROBDDs ($\phi(v) = x$):

$$v = ite(F,G,H) = (x, ite(F_x, G_x, H_x), ite(F_{\bar{x}}, G_{\bar{x}}, H_{\bar{x}}))$$

The ite Algorithm

```
struct vertex *apply_ite(struct vertex *F, *G, *H, int i)
{
  char x;
  struct vertex *η, *λ;

  if (F = v1)
    return G;
  else if (F = v0)
    return H;
  else if (G = v1 && H = v0)
    return F;
  else {
    x ← π(i);
    η ← apply_ite(Fx, Gx, Hx, i + 1);
    λ ← apply_ite(F_x̄, G_x̄, H_x̄, i + 1);
    if (η = λ)
      return η;
    else
      return old_or_new(x, η, λ);
  }
}
```

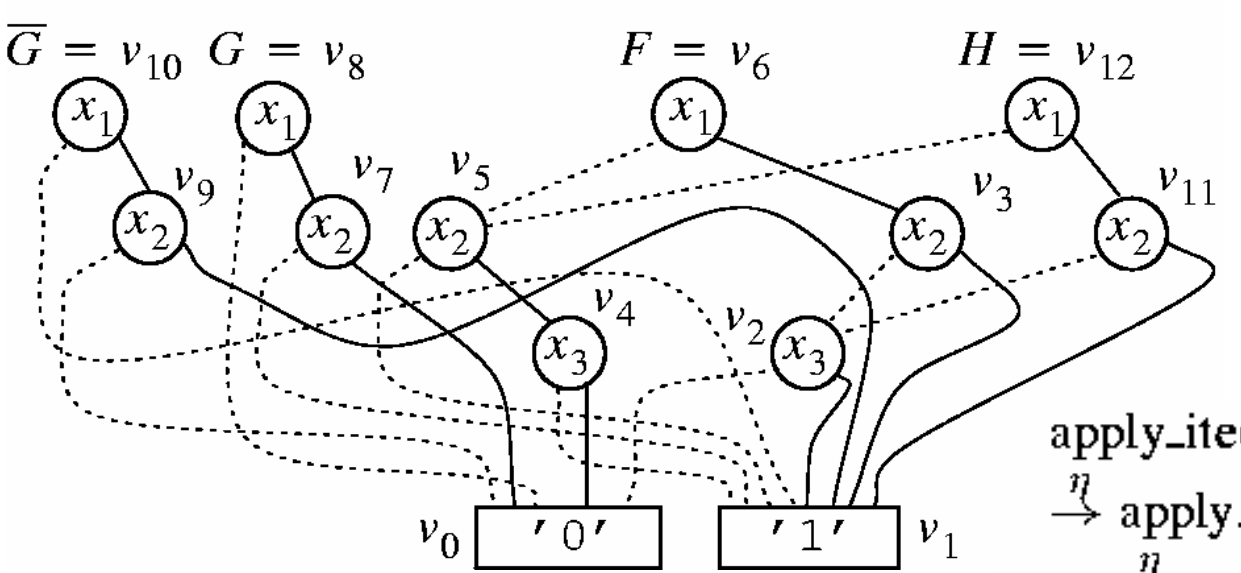
Comments on the ite Algorithm

- The algorithm processes the variables in the order used in the BDD package.
  - $\pi(i)$ gives the i^{th} variable from the top; $\pi^{-1}(x)$ gives the index position of variable x from the top.
- Computation of the restrictions: suppose that F is the root vertex of the function for which F_x should be computed:

$$F_x = \eta(F) \text{ if } \pi^{-1}(\phi(F)) = i$$

- The calculation of $F_{\overline{x}}$ is done in an analogous way.
- The time complexity of the algorithm is $O(|F|*|G|*|H|)$.

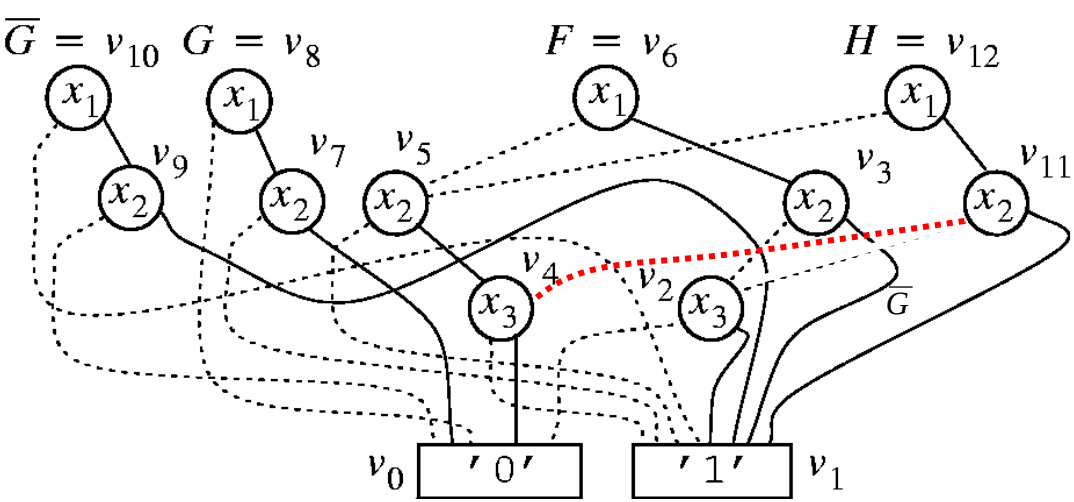
ROBDD Example: Computing $\overline{G}$ from G



$\overline{G}$ = ite(G, 0, 1)

$$
\begin{aligned}
&\text{apply\_ite}(v_8, v_0, v_1, 1) \\
&\xrightarrow{\eta} \text{apply\_ite}(v_7, v_0, v_1, 2) \\
&\quad \xrightarrow{\eta} \text{apply\_ite}(v_0, v_0, v_1, 3) \\
&\qquad v_1 \\
&\quad \xrightarrow{\lambda} \text{apply\_ite}(v_1, v_0, v_0, 3) \\
&\qquad v_0 \\
&\quad v_9 = (x_2, v_1, v_0) \\
&\xrightarrow{\lambda} \text{apply\_ite}(v_0, v_0, v_1, 2) \\
&\quad v_1 \\
&v_{10} = (x_1, v_9, v_1)
\end{aligned}
$$

ROBDD Example: Computing *H* from *F*, *G*, $\bar{G}$



$$H = F \oplus G$$
$$= ite(F, \bar{G}, G)$$

$$\begin{aligned}
&\text{apply_ite}(v_6, v_{10}, v_8, 1)\\
&\xrightarrow{\eta} \text{apply_ite}(v_3, v_9, v_7, 2)\\
&\quad\xrightarrow{\eta} \text{apply_ite}(v_1, v_1, v_0, 3)\\
&\quad\quad v_1\\
&\quad\xrightarrow{\lambda} \text{apply_ite}(v_2, v_0, v_1, 3)\\
&\quad\quad\xrightarrow{\eta} \text{apply_ite}(v_1, v_0, v_1, 4)\\
&\quad\quad\quad v_0\\
&\quad\quad\xrightarrow{\lambda} \text{apply_ite}(v_0, v_0, v_1, 4)\\
&\quad\quad\quad v_1\\
&\quad\quad v_4 = (x_3, v_0, v_1)\\
&\quad v_{11} = (x_2, v_1, v_4)\\
&\xrightarrow{\lambda} \text{apply_ite}(v_5, v_1, v_0, 2)\\
&\quad v_5\\
&v_{12} = (x_1, v_{11}, v_5)
\end{aligned}$$

Composition

- The composite problem is
  - the ROBDDs of two functions *f* and *g* are known
  - the output of *g* is connected to an input of *f*
  - compute the ROBDD of the composed function *h*, where

    $h = f(x_1, \ldots, x_{i-1}, g, x_{i+1}, \ldots, x_n)$.
- Using Shannon expansion, one finds that

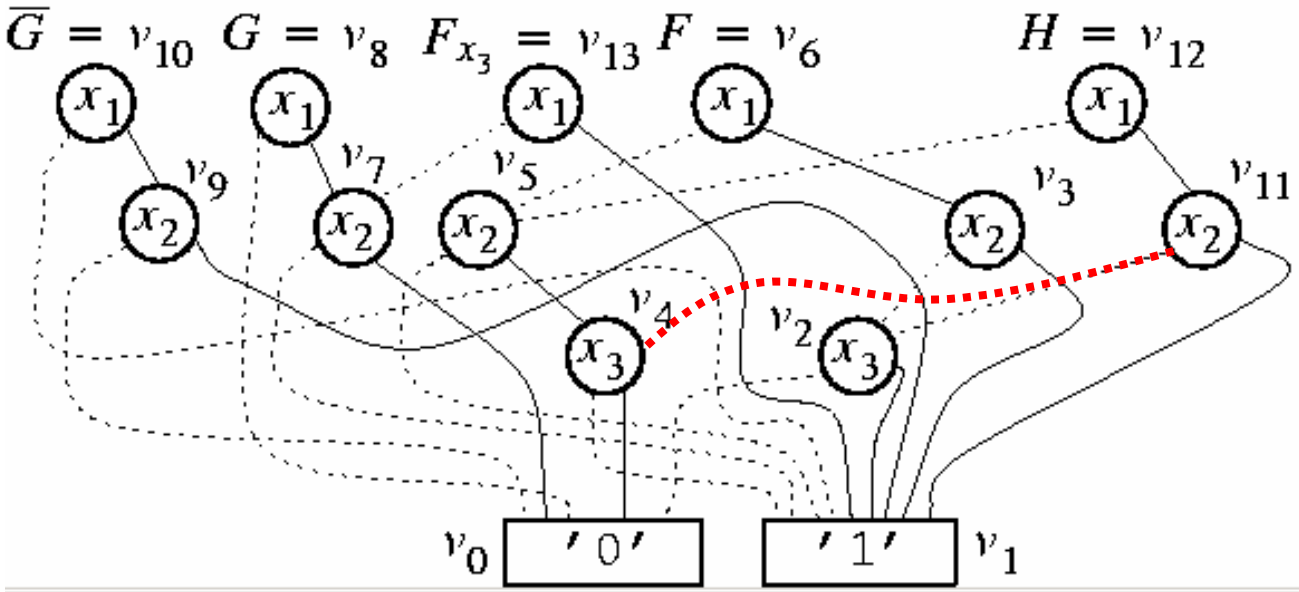
$$h = g \cdot f_{x_i} + \overline{g} \cdot f_{\overline{x_i}} = ite(g, f_{x_i}, f_{\overline{x_i}})$$

- Now, the restrictions have to be calculated by dedicated algorithms.

Positive Cofactor

```
struct vertex *positive_cofactor(struct vertex *F, int r, i)
{
  char x;
  struct vertex *η, *λ;

  if (F = v1)
    return v1;
  else if (F = v0)
    return v0;
  else if (r = i)
    return η(F);
  else {
    x ← π(i);
    η ← positive_cofactor(Fx, r, i + 1);
    λ ← positive_cofactor(Fx̄, r, i + 1);
    if (η = λ)
      return η;
    else
      return old_or_new(x, η, λ);
  }
}
```

Positive Cofactor Example: Computing F_{x3}



positive_cofactor(v_6, 3, 1)

$\xrightarrow{\eta}$ positive_cofactor(v_3, 3, 2)

$\xrightarrow{\eta}$ positive_cofactor(v_1, 3, 3)

v_1

$\xrightarrow{\lambda}$ positive_cofactor(v_2, 3, 3)

v_1

v_1

$\xrightarrow{\lambda}$ positive_cofactor(v_5, 3, 2)

$\xrightarrow{\eta}$ positive_cofactor(v_4, 3, 3)

v_0

$\xrightarrow{\lambda}$ positive_cofactor(v_1, 3, 3)

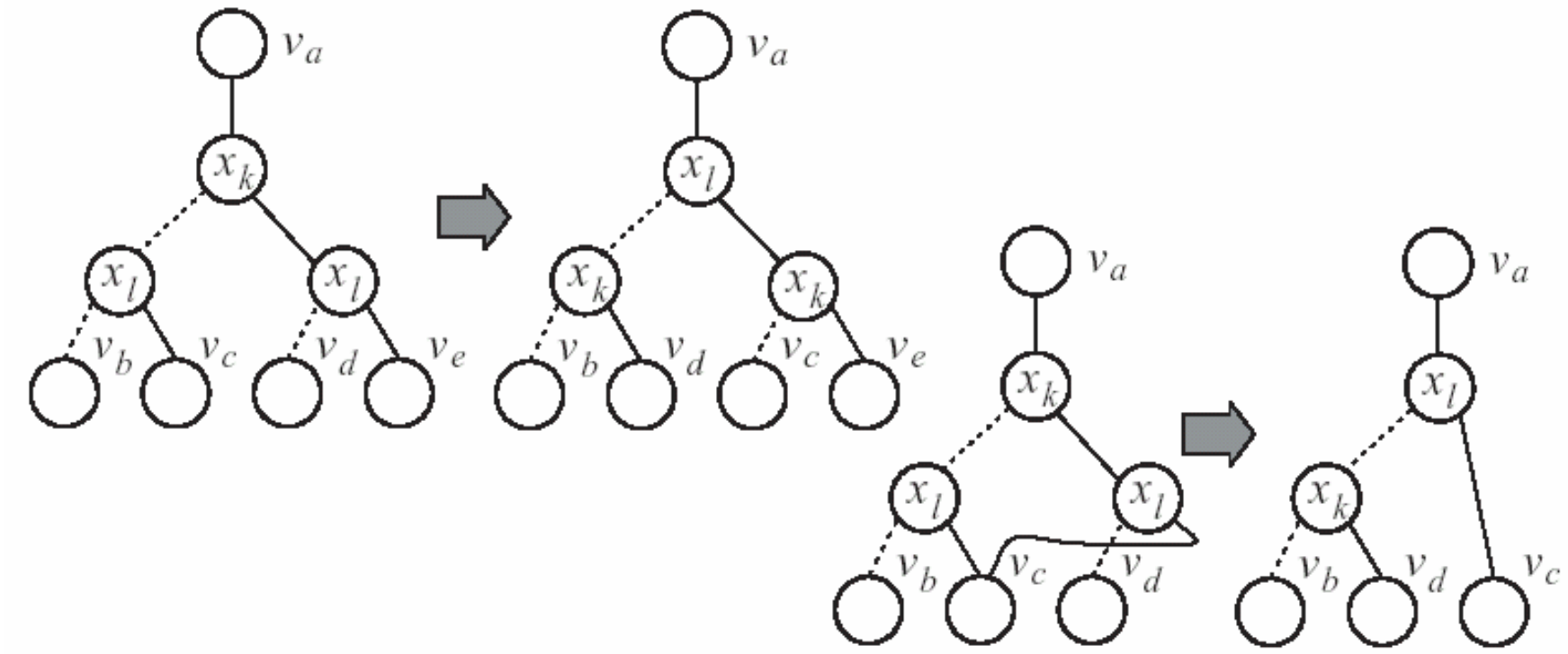
v_1

$v_7 = (x_2, v_0, v_1)$

$v_{13} = (x_1, v_1, v_7)$

Variable Ordering

- Reorder adjacent variables only has a local effect on the ROBDD.



Variable Ordering (cont'd)

- Finding the ordering that minimizes the ROBDD size for some function is intractable.
  - The optimal ordering may change as ROBDDs are being manipulated.
- So, an ROBDD package will try to reorder the variables at distinct moments.
  - It could move one variable to the top and back to the bottom and remember the best position. It could then repeat the procedure for the other variables.
- Another ‘‘invisible’’ feature of an ROBDD package is garbage collection.

The Verification Problem

- The issue is to compare a specification f to an implementation g.
- They can both be represented by ROBDDs (F resp. G).
- In case of a fully specified function, verification is trivial (pointer comparison) because of the **strong canonicity** of the ROBDD data structure.
  - Strong canonicity: the representations to identical functions are the same.
- If there is a dc-set, use two functions f and d. The implementation g is correct when $d + f \cdot g + \bar{f} \cdot \bar{g}$ is a **tautology** (the expression evaluates to '1').

ROBDDs and Satisfiability

- A Boolean function is **satisfiable** if an assignment to its variables exists for which the function becomes '1'
- Any Boolean function whose ROBDD is unequal to '0' is satisfiable.
- Suppose that choosing a Boolean variable x_i to be '1' costs c_i. Then, the **minimum-cost satisfiability** problem asks to minimize: $\sum_{i=1}^{n} c_i \mu(x_i)$

  where $\mu(x_i) = 1$ when $x_i =$ '1' and $\mu(x_i) = 0$ when $x_i =$ '0'.
- Solving minimum-cost satisfiability amounts to computing the shortest path in an ROBDD, which can be solved in linear time.
  - Weights: $w(v, \eta(v)) = c_i$, $w(v, \lambda(v)) = 0$, variable $x_i = \phi(v)$.

Applications to Combinatorial Optimization

- **Zero-one integer linear programming** can be formulated as a minimum-cost satisfiability problem.
- Consider the (standard form) constraint: $x_1 + x_2 + x_3 + x_4 = 3$.
- It can be written as:

$$(x_1 + x_2) \cdot (x_1 + x_3) \cdot (x_1 + x_4) \cdot (x_2 + x_3) \cdot (x_2 + x_4) \cdot (x_3 + x_4) \cdot (\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4)$$

  - The first 6 sums in the product: at least 3 of the 4 variables are 1.
  - The last sum: at least one of the variables is 0.
- Many combinatorial optimization problems can also be directly formulated in terms of the satisfiability problem.

Set Covering

- Given a set $S = \{s_1, \ldots, s_m\}$ and a set $K = \{K_1, \ldots, K_n\}$ where each K_j $(1 \leq j \leq n)$ is a subset of S, find a subset Γ of K such that the **union** of the elements Γ covers S.
- The cost of a cover is the sum of the costs c_j of the elements K_j of Γ.
- Multiple cost functions are possible. E.g., $c_j = 1$ or $c_j = |K_j|$.
- The problem is NP-complete for most cost functions.

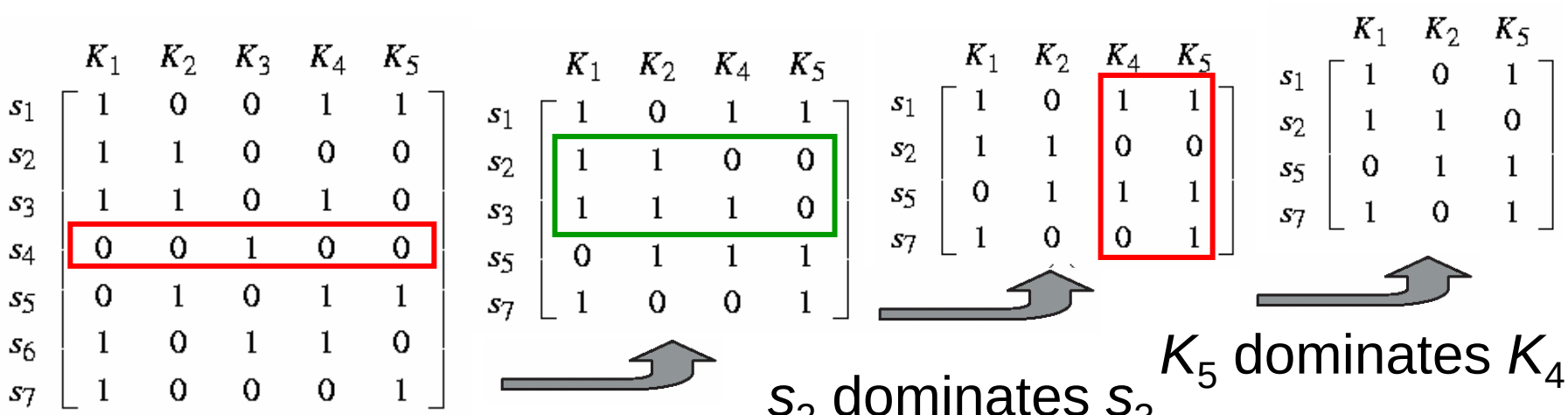
Covering Matrix

	K_1	K_2	K_3	K_4	K_5	K_6
s_1	1	1	0	0	0	1
s_2	1	0	0	1	1	1
s_3	0	1	1	0	1	0
s_4	0	1	0	1	0	1

- $\Gamma = \{K_3, K_6\}$ is the optimal solution when $c_j = |K_j|$.
- K_3 is redundant in $\Gamma = \{K_1, K_2, K_3\}$.

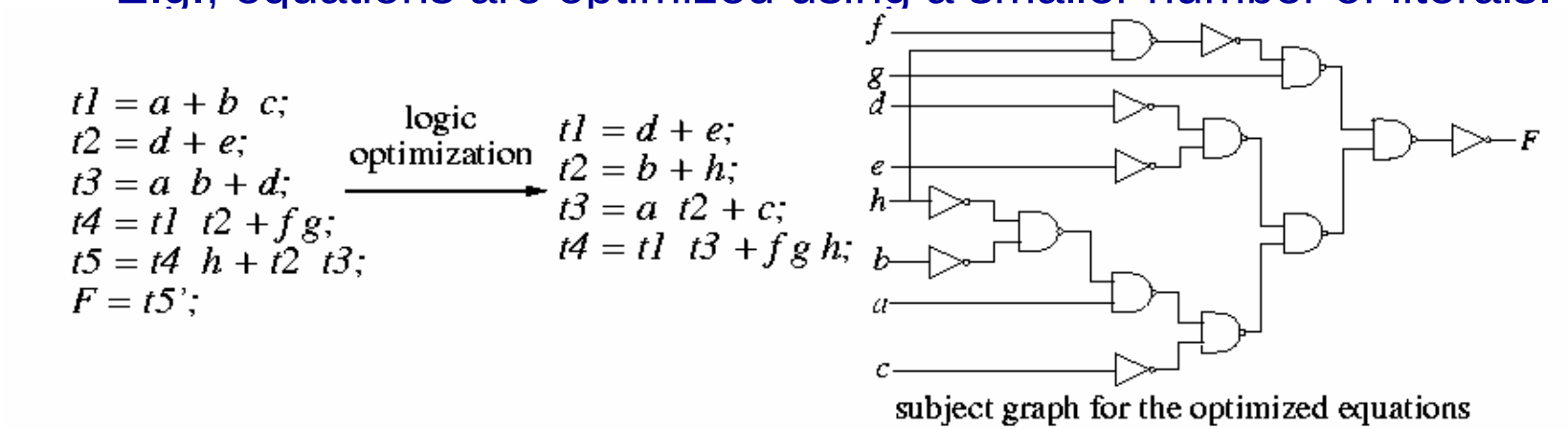
- A covering problem can be formulated as a satisfiability problem by associating variables x_j with the sets K_j: $(x_1 + x_2 + x_6) \cdot (x_1 + x_4 + x_5 + x_6) \cdot (x_2 + x_3 + x_5) \cdot (x_2 + x_4 + x_6)$.
- This type of covering is called **unate**.
- A **binate** covering problem has an expression where complemented variables are allowed.

Example Simplification Rules in Covering



Technology-Independent Logic Optimization

- **Two-level:** minimize the # of product terms.
  - $F = \bar{x_1}\bar{x_2}\bar{x_3} + \bar{x_1}\bar{x_2}x_3 + x_1\bar{x_2}\bar{x_3} + x_1\bar{x_2}x_3 + x_1x_2\bar{x_3} \Rightarrow F = \bar{x_2} + x_1\bar{x_3}$.
- **Multi-level:** minimize the #'s of literals, variables.
  - E.g., equations are optimized using a smaller number of literals.

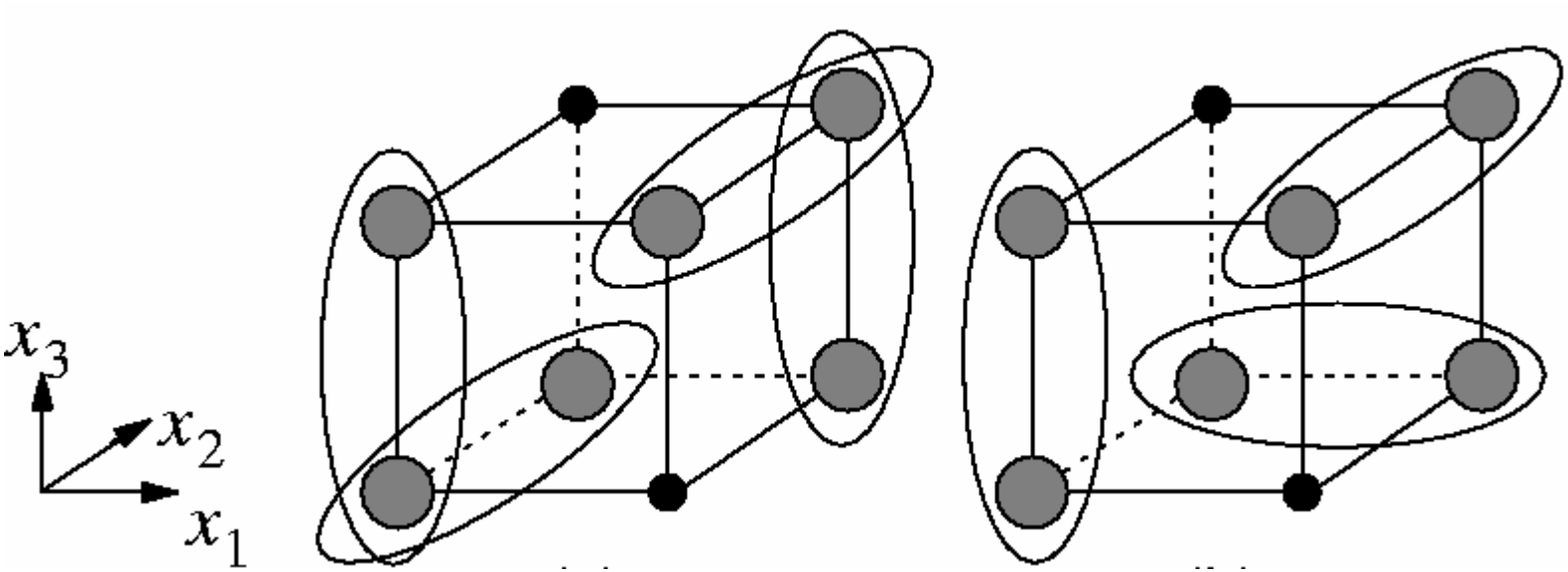


- Methods/CAD tools: The Quine-McCluskey method (exponential-time exact algorithm), Espresso (heuristics for two-level logic), MIS (heuristics for multi-level logic), Synopsys, etc.

Two-Level Logic Synthesis

- Any Boolean function can be realized in two levels: AND-OR (sum of products), NAND-NAND, etc.
- Direct implementation of two level logic using PLAs (programmable logic arrays) is not as popular as in the nMOS days.
- Classic problems, solved e.g. by the *Quine-McCluskey* algorithm.
- Popular cost function: the number of literals in the sum of products expression.
- The goal is to find a minimal irredundant prime cover.

Optimality in Two-Level Logic Synthesis



A local and a global minimum

The Quine-McCluskey Algorithm

- Calculate all prime implicants (of the union of the on-set and dc-set).
- Find the minimal cover of all minterms in the on-set by prime implicants.
- Construct the covering matrix.
- Simplify the covering matrix by detecting **essential** columns, **row and column dominance**.
- What is left is the **cyclic core** of the covering matrix.
  - The covering problem can then be solved by a branch-and-bound algorithm.
- Other methods do not first enumerate all prime implicants; they use an implicit representation by means of ROBDDs.

The Quine-McCluskey Algorithm

- $F(a, b, c, d) = \sum_m(2, 3, 7, 9, 11, 13) + \sum_d(1, 10, 15)$
- Step 1: Group minterms to find prime implicants by applying $xy + xy' = x$.
- Step 2: Select a minimum set of prime implicants (minimum # of literals) to implement the original function.
- Exponential-time exact algorithm, huge amounts of memory!

Step 1–1

1	0001	v
2	0010	v
3	0011	v
9	1001	v
10	1010	v
7	0111	v
11	1011	v
13	1101	v
15	1111	v

Step 1–2

(1, 3)	00–1	v
(1, 9)	–001	v
(2, 3)	001–	v
(2, 10)	–010	v
(3, 7)	0–11	v
(3, 11)	–011	v
(9, 11)	10–1	v
(9, 13)	1–01	v
(10, 11)	101–	v
(7, 15)	–111	v
(11, 15)	1–11	v
(13, 15)	11–1	v

Step 1–3

(1, 3, 9, 11)	–0–1
(2, 3, 10, 11)	–01–
(3, 7, 11, 15)	—11
(9, 11, 13, 15)	1—1

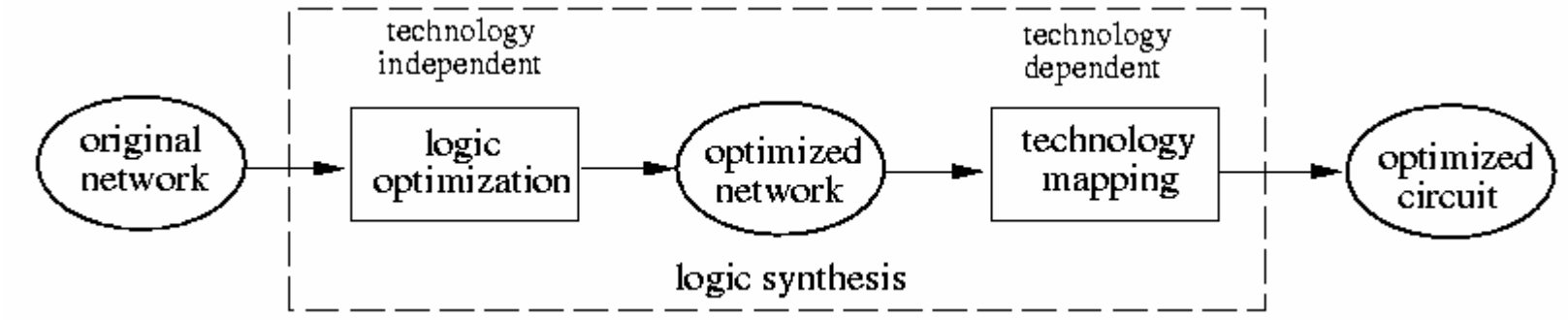
Step 2

prime imp.	2	3	7	9	11	13
(1, 3, 9, 11)						
*(2, 3, 10, 11)						
*(3, 7, 11, 15)						
*(9, 11, 13, 15)						

* essential prime implicant

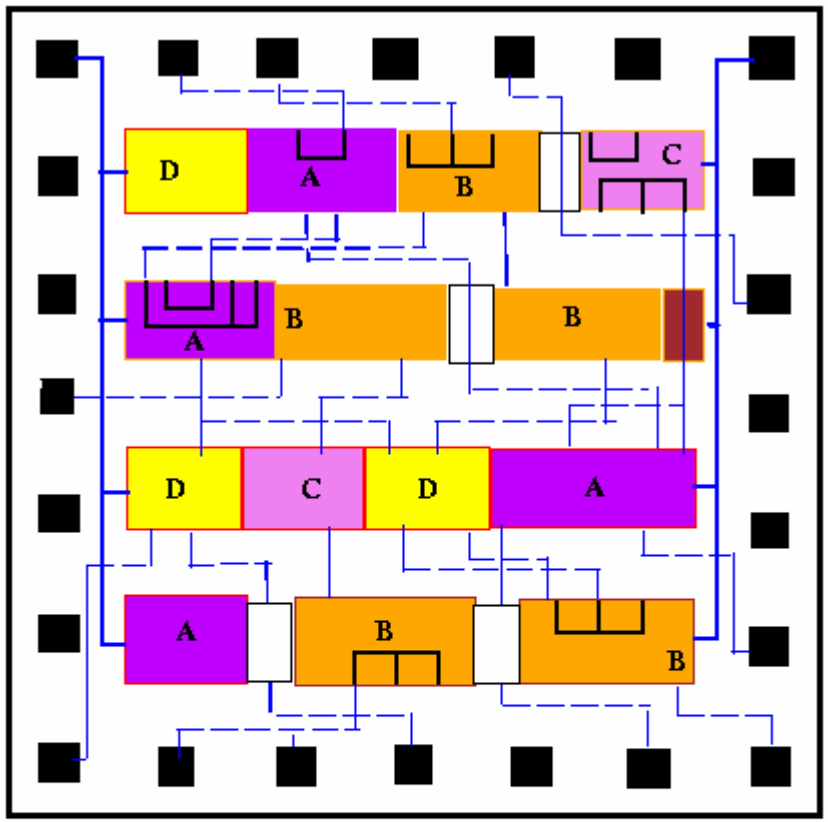
$F = b'c + cd + ad$

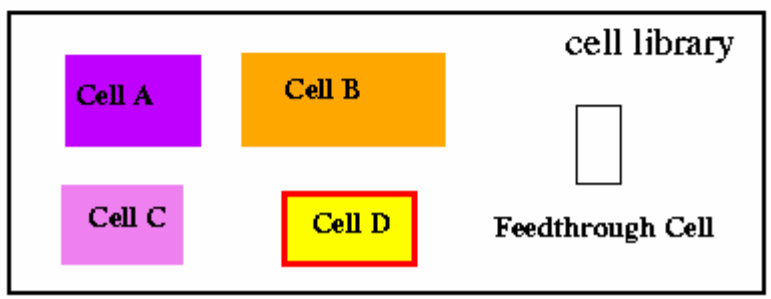
Technology Mapping



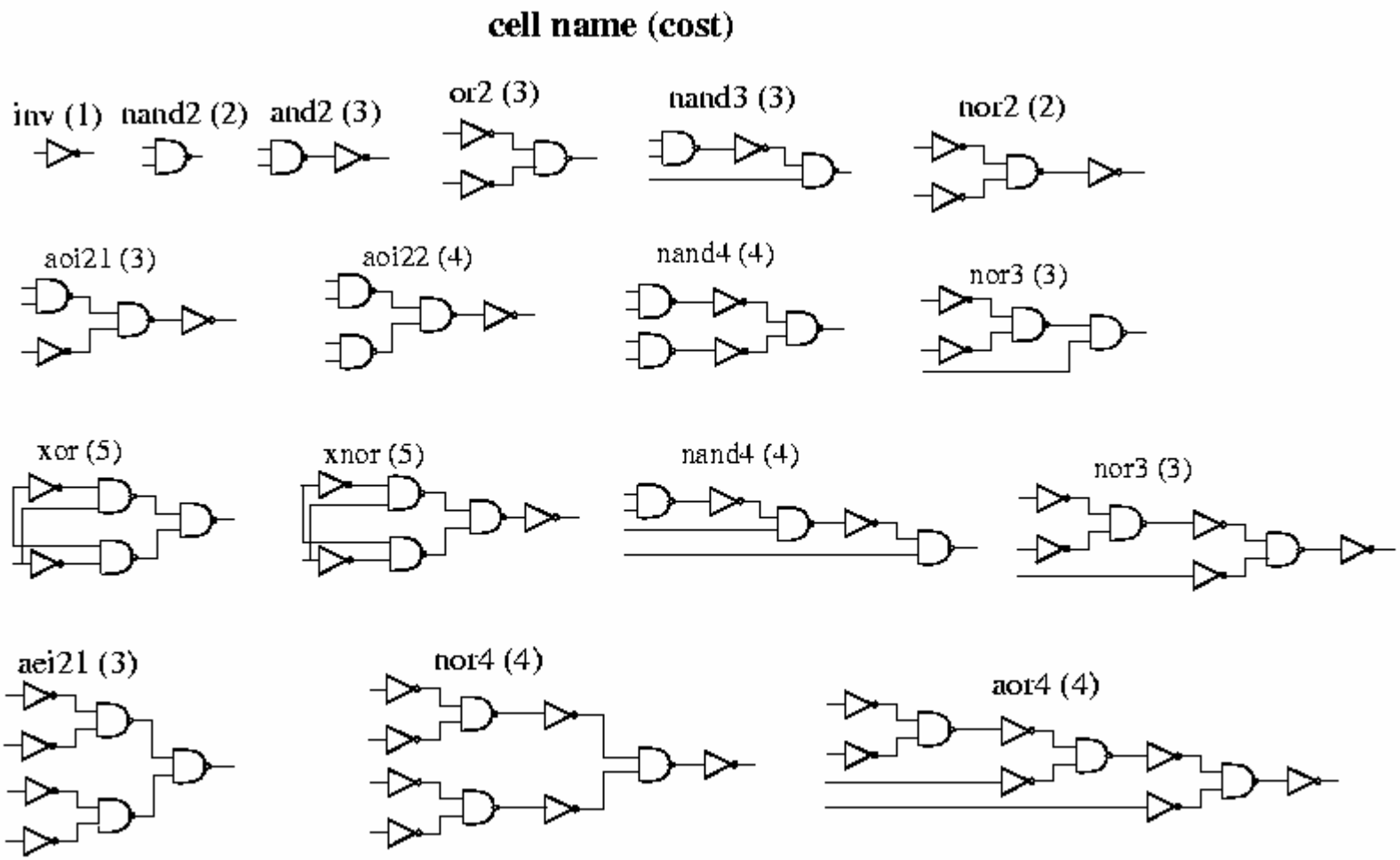
- **Library-based technology mapping:** standard cell design.
  - Map a function to a limited set of pre-designed cells
- **Lookup table-based technology mapping:** Lucent, Xilinx FPGAs, etc.
  - Each lookup table (LUT) can implement a very large number of functions (e.g., all functions with 4 inputs and 1 output)
- **Multiplexer-based technology mapping:** Actel FPGAs, etc.
  - Logic modules are constructed with multiplexers.

Standard Cell Revisited





Pattern Graphs for an Example Library



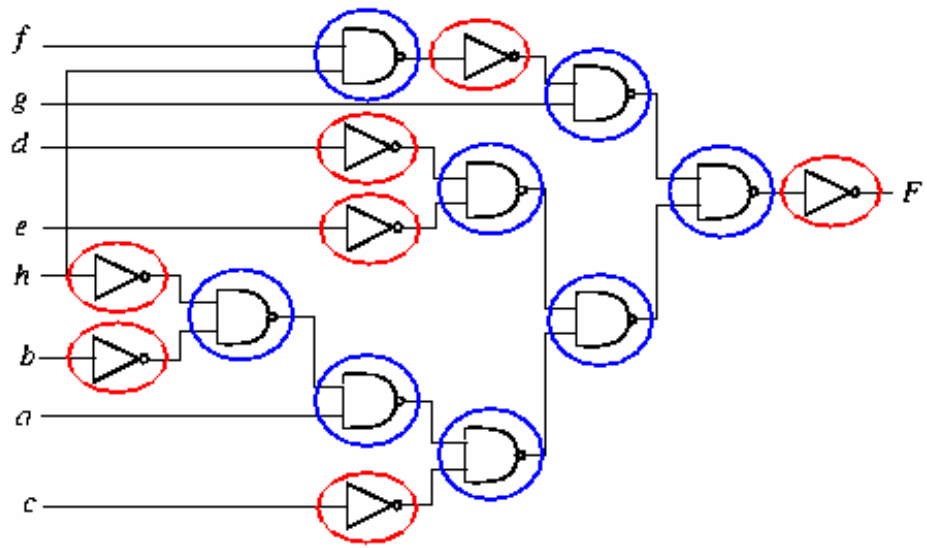
Technology Mapping

- **Technology Mapping:** The optimization problem of finding a minimum cost covering of the subject graph by choosing from the collection of pattern graphs for all gates in the library.
- A **cover** is a collection of pattern graphs such that every node of the subject graph is contained in one (or more) of the pattern graphs.
- The cover is further constrained so that each input required by a pattern graph is actually an output of some other pattern graph.

Trivial Covering

- Mapped into 2-input NANDs and 1-input inverters.
- 8 2-input NAND-gates and 7 inverters for an area cost of 23.
- Best covering?

$$t1 = d + e;$$
$$t2 = b + h;$$
$$t3 = a\ \ t2 + c;$$
$$t4 = t1\ \ t3 + f\ g\ h;$$

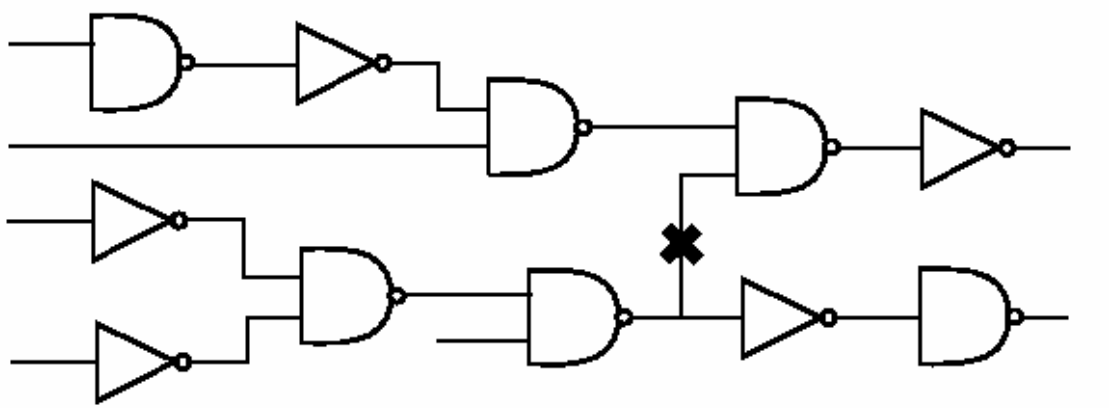


Optimal Tree Covering by Dynamic Programming

- If the subject directed acyclic graph (DAG) is a tree, then a polynomial-time algorithm to find the minimum cover exists.
  - Based on dynamic programming: optimal substructure? overlapping subproblems?
- Given: subject trees (networks to be mapped), library cells
- Consider a node *n* of the subject tree
  - Recursive assumption: For all children of *n*, a best match which implements the node is know.
  - Cost of a leaf is 0.
  - Consider each pattern tree which matches at *n*, compute cost as the cost of implementing each node which the pattern requires as an input plus the cost of the pattern.
  - Choose the lowest-cost matching pattern to implement *n*.

Tree-Covering by Dynamic Programming

- If the subject DAG is not a tree
  - Partition the subject graph into forest of trees
  - Cover each tree optimally using the dynamic programming.
  - Overall solution is only an approximation.
- Optimality
  - An optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
  - The minimum area cover for a tree *T* can be derived from the minimum area covers for every node below the root of *T*.



Best Covering

- A best covering with an area of 15.
- Obtained by the dynamic programming approach.